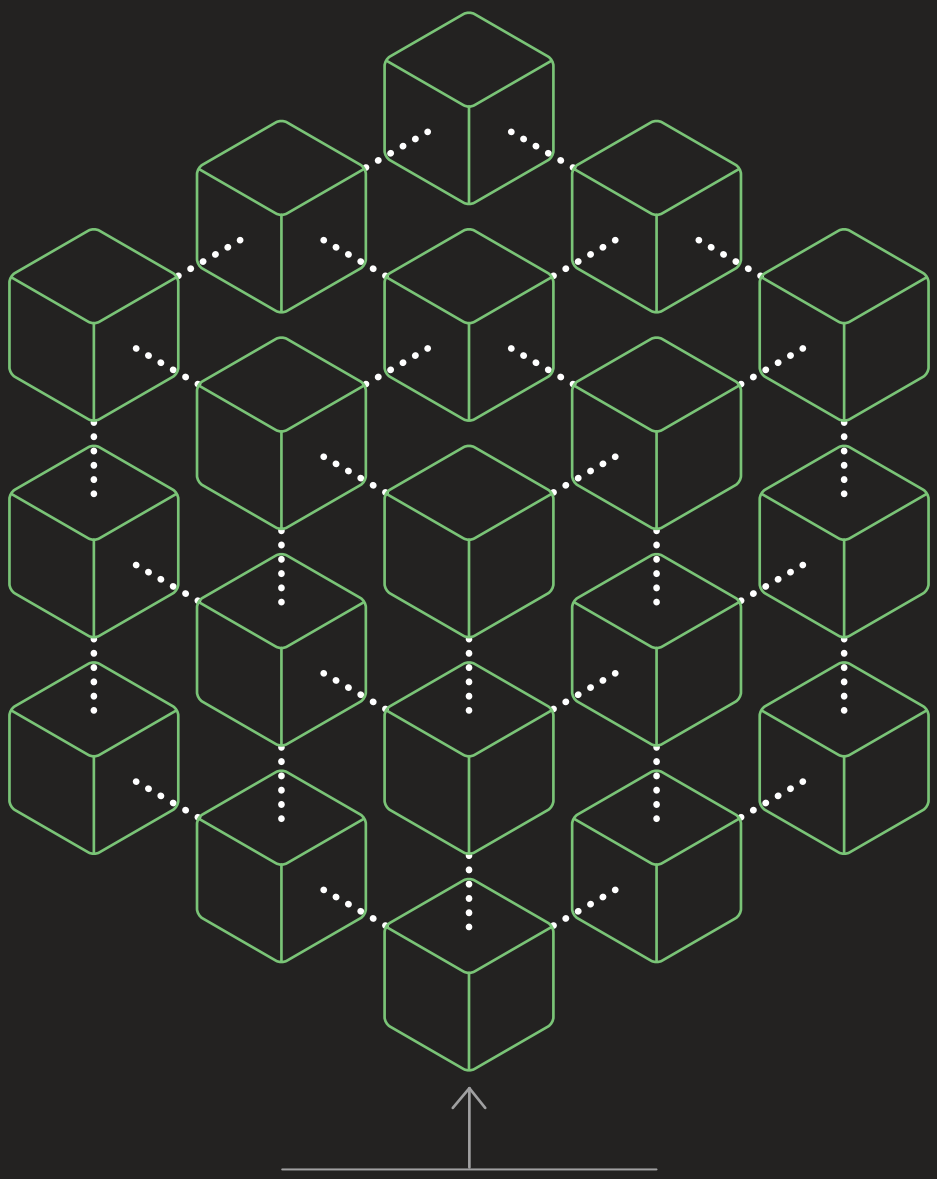
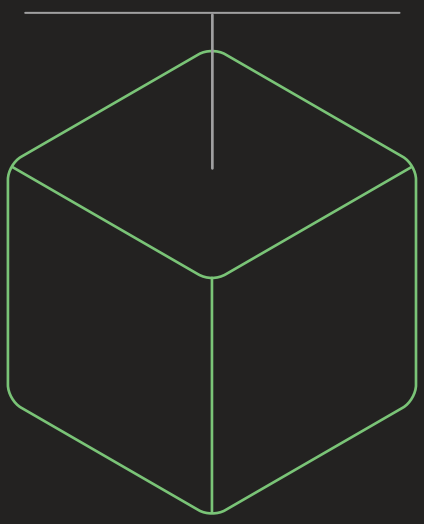


MICROSERVICES

From Design to Deployment

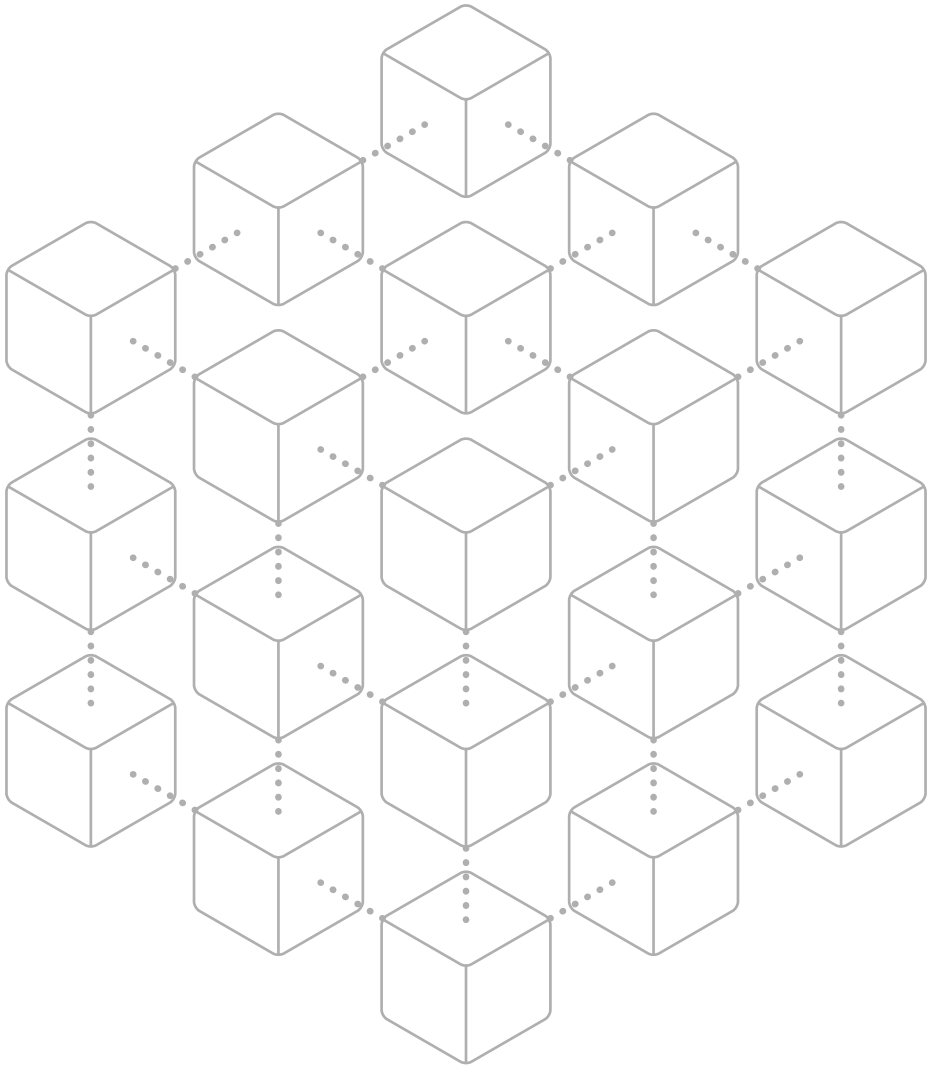


NGINX

MICROSERVICES

From Design to Deployment

by Chris Richardson
with Floyd Smith



NGINX

© NGINX, Inc. 2016

Table of Contents

Foreword iii

1 Introduction to Microservices 1
Building Monolithic Applications 1
Marching Toward Monolithic Hell 3
Microservices – Tackling the Complexity 4
The Benefits of Microservices 8
The Drawbacks of Microservices 9
Summary 11
Microservices in Action: NGINX Plus as a Reverse Proxy Server . . 11

2 Using an API Gateway 12
Introduction 12
Direct Client-to-Microservice Communication 15
Using an API Gateway 15
Benefits and Drawbacks of an API Gateway 17
Implementing an API Gateway 17
Performance and Scalability 17
Using a Reactive Programming Model 18
Service Invocation 18
Service Discovery 19
Handling Partial Failures 19
Summary 20
Microservices in Action: NGINX Plus as an API Gateway 20

3 Inter-Process Communication 21
Introduction 21
Interaction Styles 22
Defining APIs 24
Evolving APIs 24
Handling Partial Failure 25
IPC Technologies 26
Asynchronous, Message-Based Communication 26
Synchronous, Request/Response IPC 29
REST 29
Thrift 31
Message Formats 31
Summary 32
Microservices in Action: NGINX and Application Architecture . . . 33

4 Service Discovery 34
Why Use Service Discovery? 34
The Client-Side Discovery Pattern 35
The Server-Side Discovery Pattern 37
The Service Registry 38
Service Registration Options 39
The Self-Registration Pattern 39
The Third-Party Registration Pattern 41
Summary 42
Microservices in Action: NGINX Flexibility 43

5 Event-Driven Data Management for Microservices 44
Microservices and the Problem of Distributed Data Management 44
Event-Driven Architecture 47
Achieving Atomicity 50
Publishing Events Using Local Transactions 50
Mining a Database Transaction Log 51
Using Event Sourcing 52
Summary 54
Microservices in Action: NGINX and Storage Optimization 54

6 Choosing a Microservices Deployment Strategy 55
Motivations 55
Multiple Service Instances per Host Pattern 56
Service Instance per Host Pattern 58
Service Instance per Virtual Machine Pattern 58
Service Instance per Container Pattern 60
Serverless Deployment 62
Summary 63
Microservices in Action: Deploying Microservices Across Varying Hosts with NGINX 63

7 Refactoring a Monolith into Microservices 64
Overview of Refactoring to Microservices 65
Strategy #1: Stop Digging 66
Strategy #2: Split Frontend and Backend 67
Strategy #3: Extract Services 69
Prioritizing Which Modules to Convert into Services 69
How to Extract a Module 69
Summary 71
Microservices in Action: Taming a Monolith with NGINX 72

Resources for Microservices and NGINX 73

Foreword

by Floyd Smith

The rise of microservices has been a remarkable advancement in application development and deployment. With microservices, an application is developed, or refactored, into separate services that "speak" to one another in a well-defined way – via APIs, for instance. Each microservice is self-contained, each maintains its own data store (which has significant implications), and each can be updated independently of others.

Moving to a microservices-based approach makes app development faster and easier to manage, requiring fewer people to implement more new features. Changes can be made and deployed faster and easier. An application designed as a collection of microservices is easier to run on multiple servers with load balancing, making it easy to handle demand spikes and steady increases in demand over time, while reducing downtime caused by hardware or software problems.

Microservices are a critical part of a number of significant advancements that are changing the nature of how we work. Agile software development techniques, moving applications to the cloud, DevOps culture, continuous integration and continuous deployment (CI/CD), and the use of containers are all being used alongside microservices to revolutionize application development and delivery.

NGINX software is strongly associated with microservices and all of the technologies listed above. Whether deployed as a reverse proxy, or as a highly efficient web server, NGINX makes microservices-based application development easier and keeps microservices-based solutions running smoothly.

With the tie between NGINX and microservices being so strong, we've run a seven-part series on microservices on the NGINX website. Written by Chris Richardson, who has had early involvement with the concept and its implementation, the blog posts cover the major aspects of microservices for app design and development, including how to make the move from a monolithic application. The blog posts offer a thorough overview of major microservices issues and have been extremely popular.

In this ebook, we've converted each blog post to a book chapter, and added a sidebar to each chapter with information relevant to implementing microservices in NGINX. If you follow the advice herein carefully, you'll solve many potential development problems before you even start writing code. This book is also a good companion to the NGINX Microservices Reference Architecture, which implements much of the theory presented here.

The book chapters are:

1. **Introduction to Microservices** – A clear and simple introduction to microservices, from its perhaps overhyped conceptual definition to the reality of how microservices are deployed in creating and maintaining applications.
2. **Using an API Gateway** – An API Gateway is the single point of entry for your entire microservices-based application, presenting the API for each microservice. NGINX Plus can effectively be used as an API Gateway with load balancing, static file caching, and more.
3. **Inter-process Communication in a Microservices Architecture** – Once you break a monolithic application into separate pieces – microservices – the pieces need to speak to each other. And it turns out that you have many options for inter-process communication, including representational state transfer (REST). This chapter gives the details.
4. **Service Discovery in a Microservices Architecture** – When services are running in a dynamic environment, finding them when you need them is not a trivial issue. In this chapter, Chris describes a practical solution to this problem.
5. **Event-Driven Data Management for Microservices** – Instead of sharing a unified application-wide data store (or two) across a monolithic application, each microservice maintains its own unique data representation and storage. This gives you great flexibility, but can also cause complexity, and this chapter helps you sort through it.
6. **Choosing a Microservices Deployment Strategy** – In a DevOps world, how you do things is just as important as what you set out to do in the first place. Chris describes the major patterns for microservices deployment so you can make an informed choice for your own application.
7. **Refactoring a Monolith into Microservices** – In a perfect world, we would always get the time and money to convert core software into the latest and greatest technologies, tools, and approaches, with no real deadlines. But you may well find yourself converting a monolith into microservices, one… small… piece… at… a… time. Chris presents a strategy for doing this sensibly.

We think you'll find every chapter worthwhile, and we hope that you'll come back to this ebook as you develop your own microservices apps.

Floyd Smith
NGINX, Inc.

1 Introduction to Microservices

Microservices are currently getting a lot of attention: articles, blogs, discussions on social media, and conference presentations. They are rapidly heading towards the peak of inflated expectations on the Gartner Hype cycle. At the same time, there are skeptics in the software community who dismiss microservices as nothing new. Naysayers claim that the idea is just a rebranding of service-oriented architecture (SOA). However, despite both the hype and the skepticism, the Microservices Architecture pattern has significant benefits – especially when it comes to enabling the agile development and delivery of complex enterprise applications.

This chapter is the first in this seven-chapter ebook about designing, building, and deploying microservices. You will learn about the microservices approach and how it compares to the more traditional Monolithic Architecture pattern. This ebook will describe the various elements of a microservices architecture. You will learn about the benefits and drawbacks of the Microservices Architecture pattern, whether it makes sense for your project, and how to apply it.

Let's first look at why you should consider using microservices.

Building Monolithic Applications

Let's imagine that you were starting to build a brand new taxi-hailing application intended to compete with Uber and Hailo. After some preliminary meetings and requirements gathering, you would create a new project either manually or by using a generator that comes with a platform such as Rails, Spring Boot, Play, or Maven.

This new application would have a modular hexagonal architecture, like in Figure 1-1:

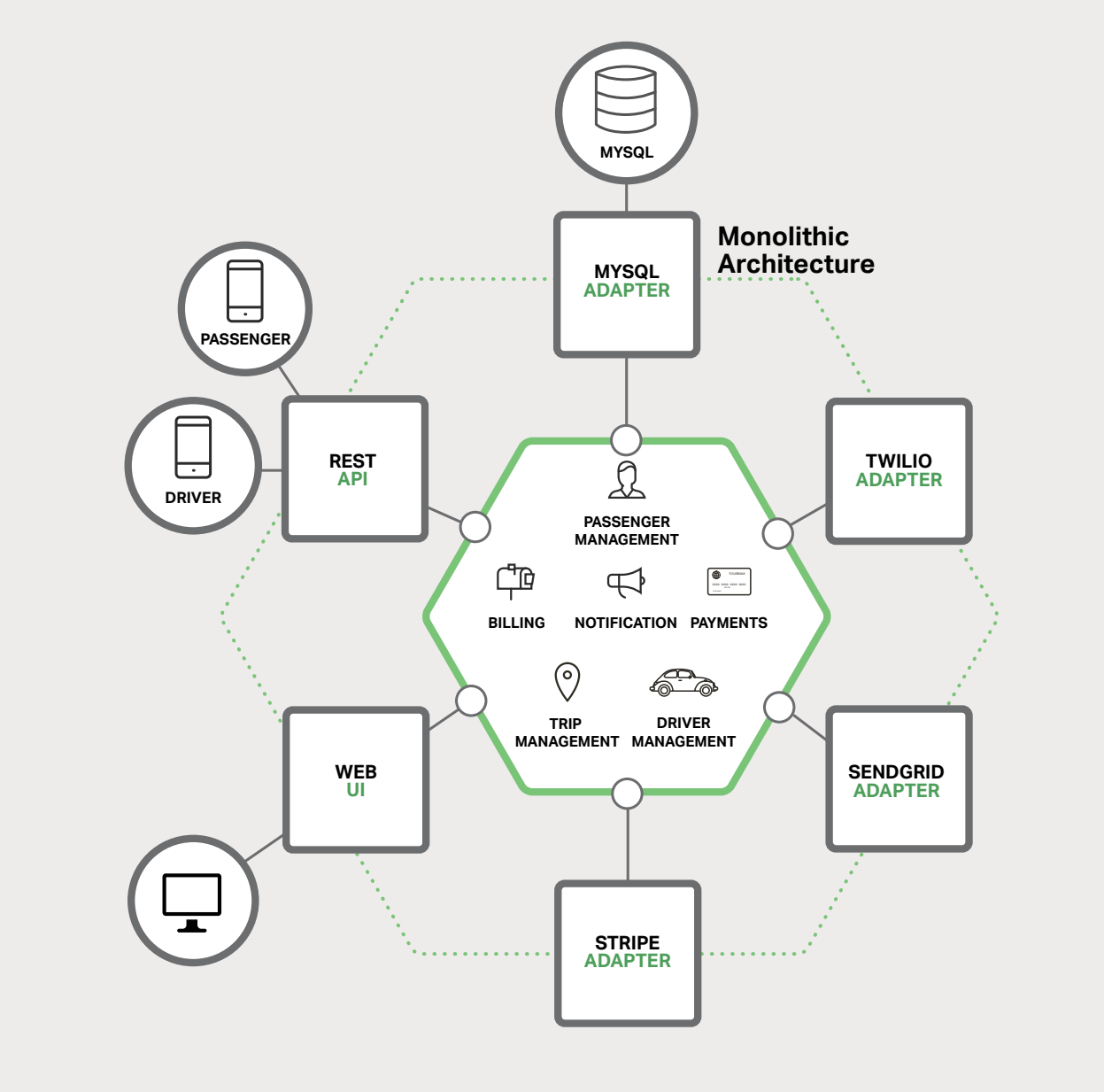


Figure 1-1. A sample taxi-hailing application.

At the core of the application is the business logic, which is implemented by modules that define services, domain objects, and events. Surrounding the core are adapters that interface with the external world. Examples of adapters include database access components, messaging components that produce and consume messages, and web components that either expose APIs or implement a UI.

Despite having a logically modular architecture, the application is packaged and deployed as a monolith. The actual format depends on the application's language and framework. For example, many Java applications are packaged as WAR files and deployed on application servers such as Tomcat or Jetty. Other Java applications are packaged as self-contained executable JARs. Similarly, Rails and Node.js applications are packaged as a directory hierarchy.

Applications written in this style are extremely common. They are simple to develop since our IDEs and other tools are focused on building a single application. These kinds of applications are also simple to test. You can implement end-to-end testing by simply launching the application and testing the UI with a testing package such as Selenium. Monolithic applications are also simple to deploy. You just have to copy the packaged application to a server. You can also scale the application by running multiple copies behind a load balancer. In the early stages of the project it works well.

Marching Toward Monolithic Hell

Unfortunately, this simple approach has a huge limitation. Successful applications have a habit of growing over time and eventually becoming huge. During each sprint, your development team implements a few more user stories, which, of course, means adding many lines of code. After a few years, your small, simple application will have grown into a monstrous monolith. To give an extreme example, I recently spoke to a developer who was writing a tool to analyze the dependencies between the thousands of JARs in their multi-million lines of code (LOC) application. I'm sure it took the concerted effort of a large number of developers over many years to create such a beast.

Once your application has become a large, complex monolith, your development organization is probably in a world of pain. Any attempts at agile development and delivery will flounder. One major problem is that the application is overwhelmingly complex. It's simply too large for any single developer to fully understand. As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming. What's more, this tends to be a downwards spiral. If the codebase is difficult to understand, then changes won't be made correctly. You will end up with a monstrous, incomprehensible big ball of mud.

The sheer size of the application will also slow down development. The larger the application, the longer the start-up time is. I surveyed developers about the size and performance of their monolithic applications, and some reported start-up times as long as 12 minutes. I've also heard anecdotes of applications taking as long as 40 minutes to start up. If developers regularly have to restart the application server, then a large part of their day will be spent waiting around and their productivity will suffer.

Another problem with a large, complex monolithic application is that it is an obstacle to continuous deployment. Today, the state of the art for SaaS applications is to push changes into production many times a day. This is extremely difficult to do with a complex monolith,

since you must redeploy the entire application in order to update any one part of it. The lengthy start-up times that I mentioned earlier won't help either. Also, since the impact of a change is usually not very well understood, it is likely that you have to do extensive manual testing. Consequently, continuous deployment is next to impossible to do.

Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements. For example, one module might implement CPU-intensive image processing logic and would ideally be deployed in Amazon EC2 Compute Optimized instances. Another module might be an in-memory database and best suited for EC2 Memory-optimized instances. However, because these modules are deployed together, you have to compromise on the choice of hardware.

Another problem with monolithic applications is reliability. Because all modules are running within the same process, a bug in any module, such as a memory leak, can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.

Last but not least, monolithic applications make it extremely difficult to adopt new frameworks and languages. For example, let's imagine that you have 2 million lines of code written using the XYZ framework. It would be extremely expensive (in both time and cost) to rewrite the entire application to use the newer ABC framework, even if that framework was considerably better. As a result, there is a huge barrier to adopting new technologies. You are stuck with whatever technology choices you made at the start of the project.

To summarize: you have a successful business-critical application that has grown into a monstrous monolith that very few, if any, developers understand. It is written using obsolete, unproductive technology that makes hiring talented developers difficult. The application is difficult to scale and is unreliable. As a result, agile development and delivery of applications is impossible.

So what can you do about it?

Microservices – Tackling the Complexity

Many organizations, such as Amazon, eBay, and Netflix, have solved this problem by adopting what is now known as the Microservices Architecture pattern. Instead of building a single monstrous, monolithic application, the idea is to split your application into set of smaller, interconnected services.

A service typically implements a set of distinct features or functionality, such as order management, customer management, etc. Each microservice is a mini-application that has its own hexagonal architecture consisting of business logic along with various adapters. Some microservices would expose an API that's consumed by other microservices or by the application's clients. Other microservices might implement a web UI. At runtime, each instance is often a cloud virtual machine (VM) or a Docker container.

For example, a possible decomposition of the system described earlier is shown in Figure 1-2:

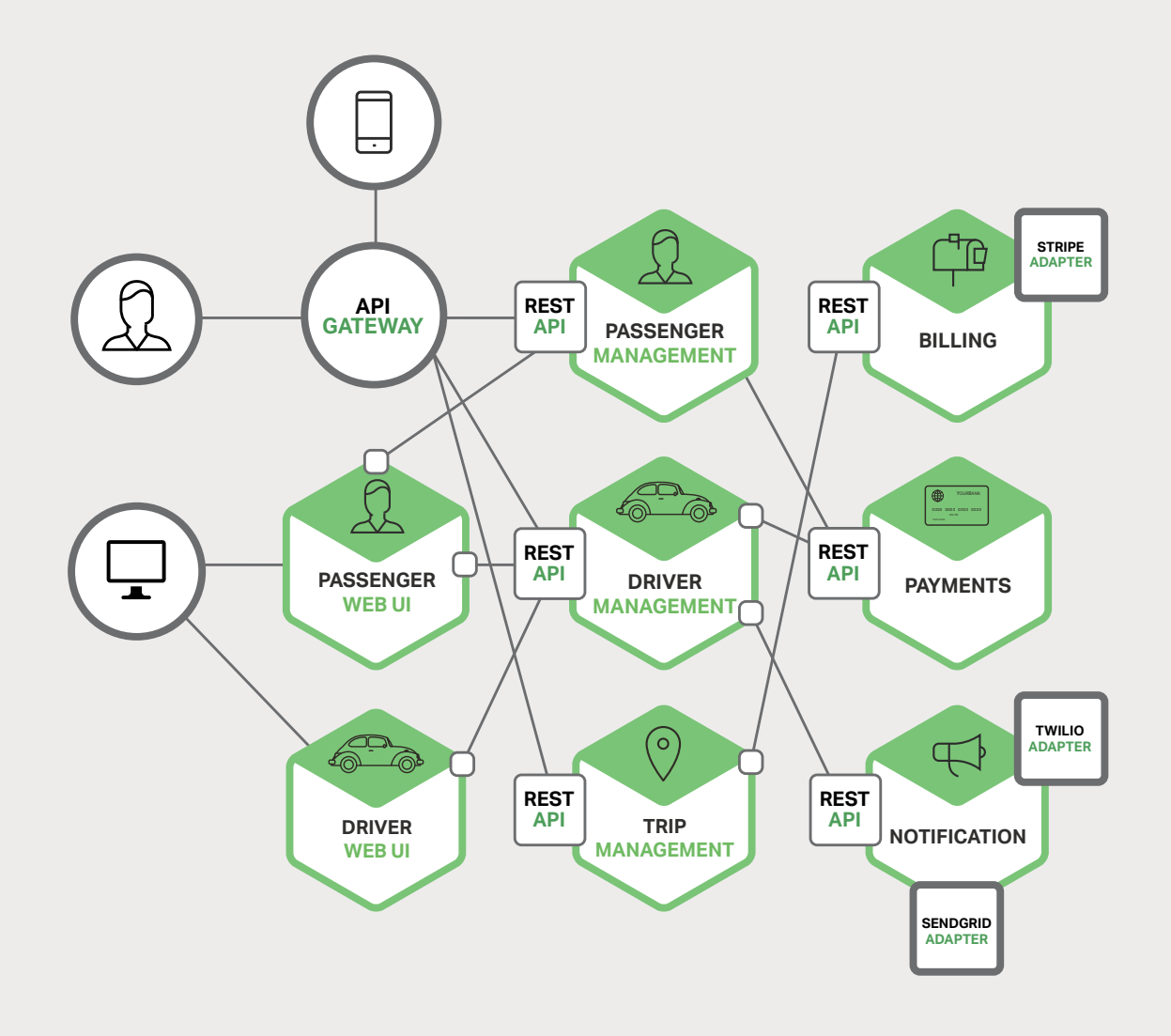


Figure 1-2. A monolithic application decomposed into microservices.

Each functional area of the application is now implemented by its own microservice. Moreover, the web application is split into a set of simpler web applications – such as one for passengers and one for drivers, in our taxi-hailing example. This makes it easier to deploy distinct experiences for specific users, devices, or specialized use cases.

Each backend service exposes a REST API and most services consume APIs provided by other services. For example, Driver Management uses the Notification server to tell an available driver about a potential trip. The UI services invoke the other services in order to render web pages. Services might also use asynchronous, message-based communication. Inter-service communication will be covered in more detail later in this ebook.

Some REST APIs are also exposed to the mobile apps used by the drivers and passengers. The apps don't, however, have direct access to the backend services. Instead, communication is mediated by an intermediary known as an API Gateway. The API Gateway is responsible for tasks such as load balancing, caching, access control, API metering, and monitoring, and can be implemented effectively using NGINX. Chapter 2 discusses the API Gateway in detail.

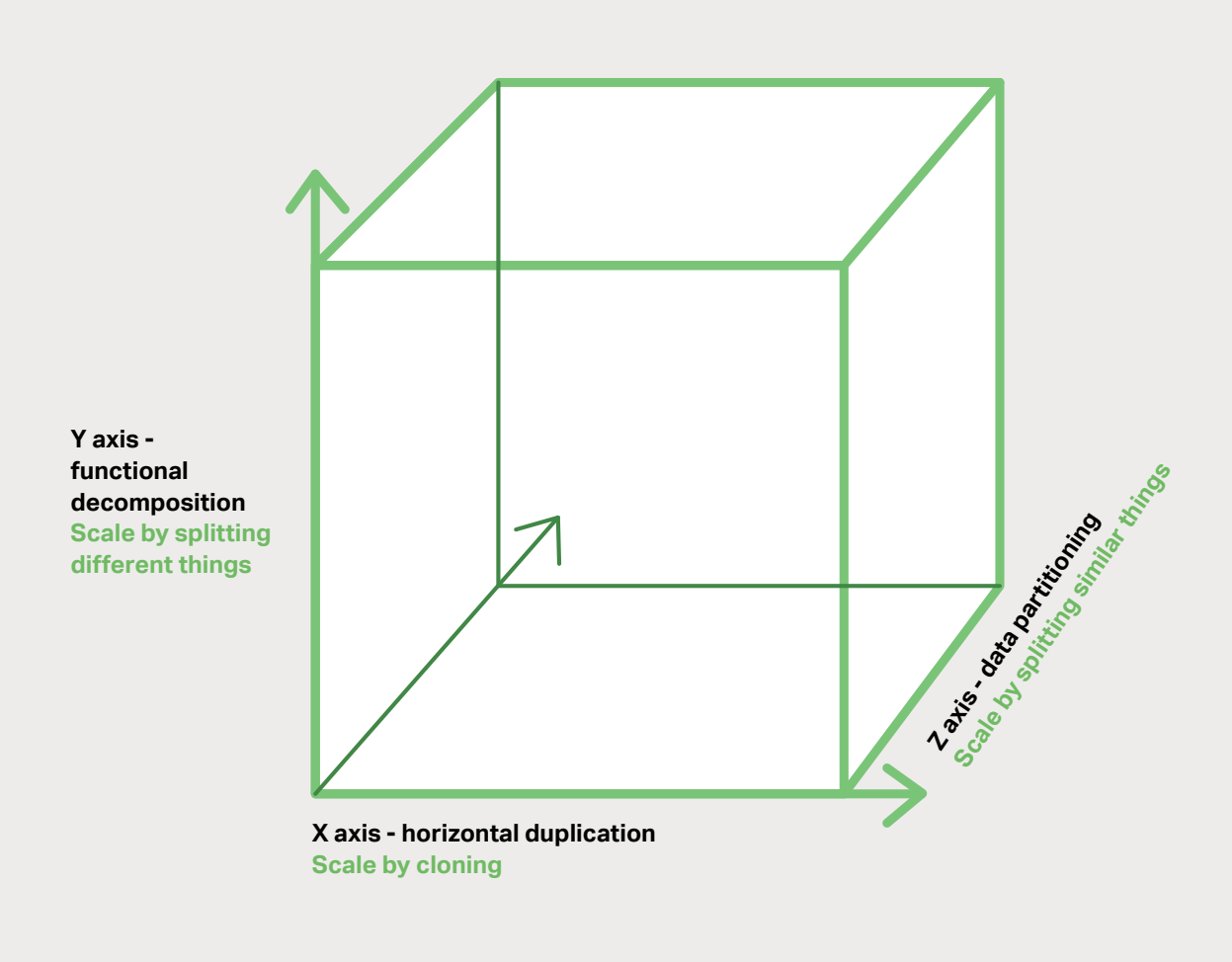


Figure 1-3. The Scale Cube, used in both development and delivery.

The Microservices Architecture pattern corresponds to the Y-axis scaling of the Scale Cube, which is a 3D model of scalability from the excellent book *The Art of Scalability*. The other two scaling axes are X-axis scaling, which consists of running multiple identical copies of the application behind a load balancer, and Z-axis scaling (or data partitioning), where an attribute of the request (for example, the primary key of a row or identity of a customer) is used to route the request to a particular server.

Applications typically use the three types of scaling together. Y-axis scaling decomposes the application into microservices as shown above in Figure 1-2.

At runtime, X-axis scaling runs multiple instances of each service behind a load balancer for throughput and availability. Some applications might also use Z-axis scaling to partition the services. Figure 1-4 shows how the Trip Management service might be deployed with Docker running on Amazon EC2.

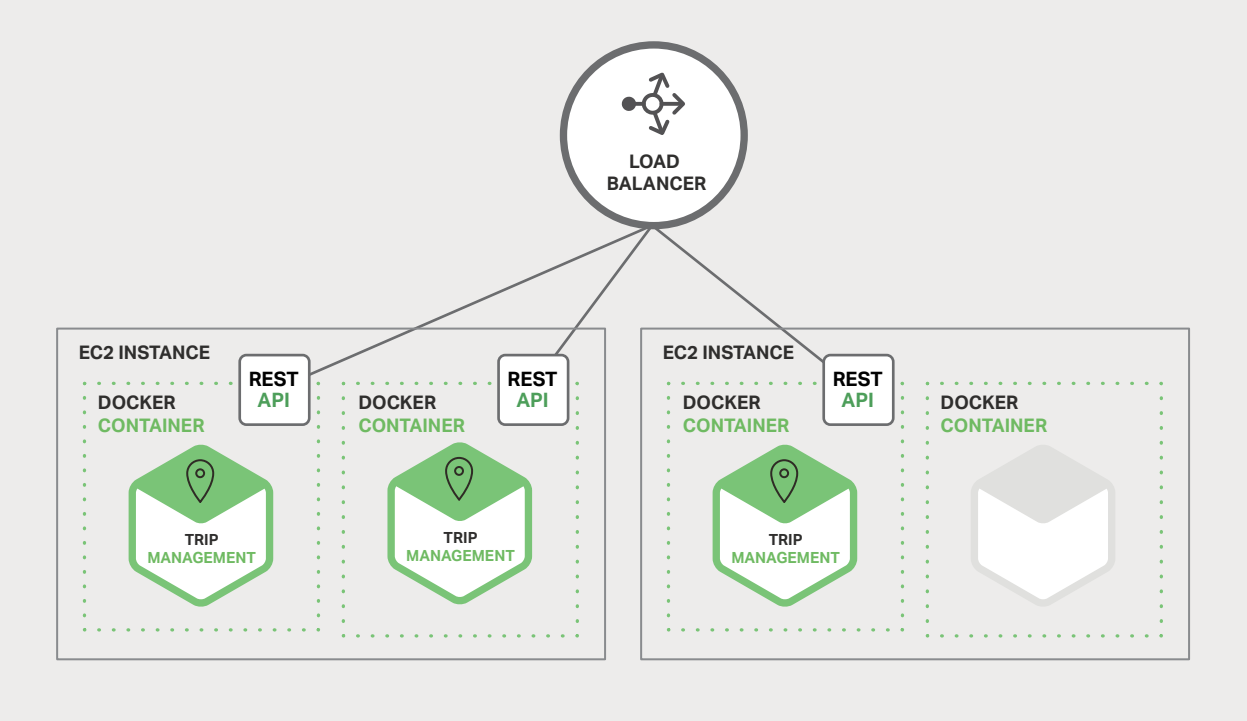


Figure 1-4. Deploying the Trip Management service using Docker.

At runtime, the Trip Management service consists of multiple service instances. Each service instance is a Docker container. In order to be highly available, the containers are running on multiple Cloud VMs. In front of the service instances is a load balancer such as NGINX that distributes requests across the instances. The load balancer might also handle other concerns such as caching, access control, API metering, and monitoring.

The Microservices Architecture pattern significantly impacts the relationship between the application and the database. Rather than sharing a single database schema with other services, each service has its own database schema. On the one hand, this approach is at odds with the idea of an enterprise-wide data model. Also, it often results in duplication of some data. However, having a database schema per service is essential if you want to benefit from microservices, because it ensures loose coupling. Figure 1-5 shows the database architecture for the sample application.

Each of the services has its own database. Moreover, a service can use a type of database that is best suited to its needs, the so-called polyglot persistence architecture. For example, Driver Management, which finds drivers close to a potential passenger, must use a database that supports efficient geo-queries.

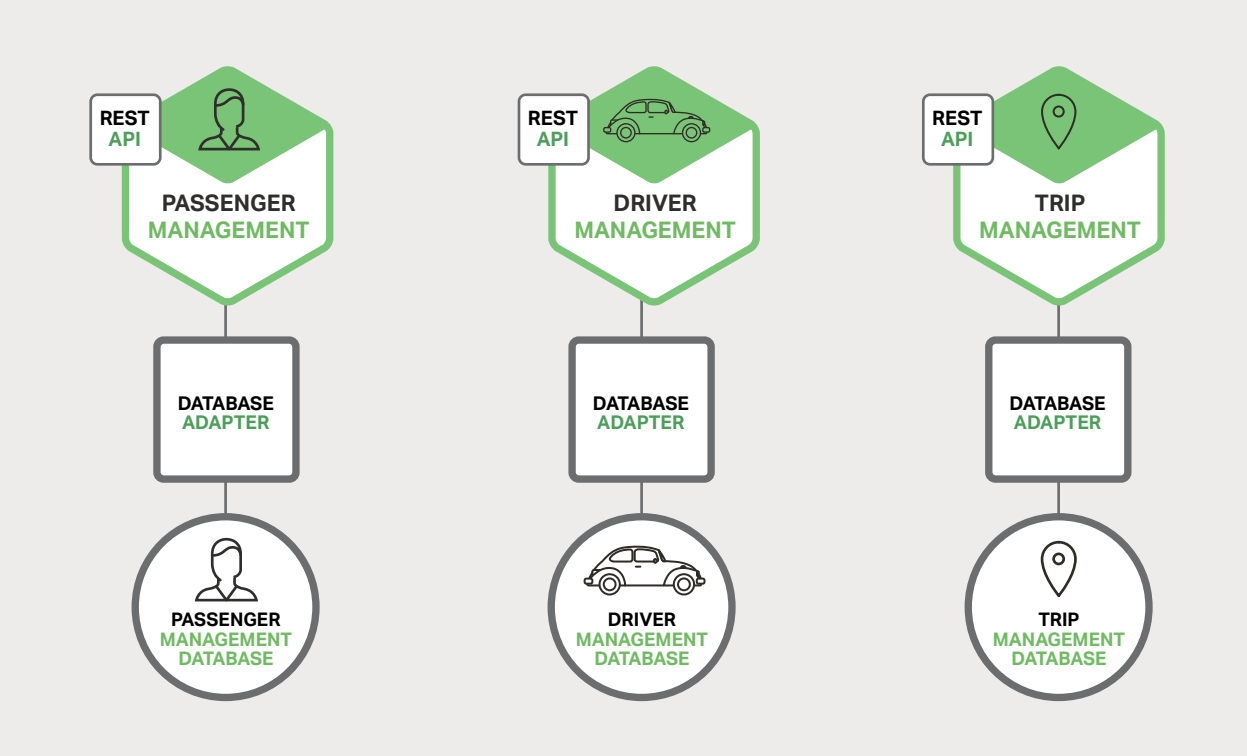


Figure 1-5. Database architecture for the taxi-hailing application.

On the surface, the Microservices Architecture pattern is similar to SOA. With both approaches, the architecture consists of a set of services. However, one way to think about the Microservices Architecture pattern is that it's SOA without the commercialization and perceived baggage of web service specifications (WS-*) and an Enterprise Service Bus (ESB). Microservice-based applications favor simpler, lightweight protocols such as REST, rather than WS-*. They also very much avoid using ESBs and instead implement ESB-like functionality in the microservices themselves. The Microservices Architecture pattern also rejects other parts of SOA, such as the concept of a canonical schema for data access.

The Benefits of Microservices

The Microservices Architecture pattern has a number of important benefits. First, it tackles the problem of complexity. It decomposes what would otherwise be a monstrous monolithic application into a set of services. While the total amount of functionality is unchanged, the application has been broken up into manageable chunks or services. Each service has a well-defined boundary in the form of a remote procedure call (RPC)-driven or message-driven API. The Microservices Architecture pattern enforces a level of modularity that in practice is extremely difficult to achieve with a monolithic code base. Consequently, individual services are much faster to develop, and much easier to understand and maintain.

Second, this architecture enables each service to be developed independently by a team that is focused on that service. The developers are free to choose whatever technologies make sense, provided that the service honors the API contract. Of course, most organizations would want to avoid complete anarchy by limiting technology options. However, this freedom means that developers are no longer obligated to use the possibly obsolete technologies that existed at the start of a new project. When writing a new service, they have the option of using current technology. Moreover, since services are relatively small, it becomes more feasible to rewrite an old service using current technology.

Third, the Microservices Architecture pattern enables each microservice to be deployed independently. Developers never need to coordinate the deployment of changes that are local to their service. These kinds of changes can be deployed as soon as they have been tested. The UI team can, for example, perform A|B testing and rapidly iterate on UI changes. The Microservices Architecture pattern makes continuous deployment possible.

Finally, the Microservices Architecture pattern enables each service to be scaled independently. You can deploy just the number of instances of each service that satisfy its capacity and availability constraints. Moreover, you can use the hardware that best matches a service's resource requirements. For example, you can deploy a CPU-intensive image processing service on EC2 Compute Optimized instances and deploy an in-memory database service on EC2 Memory-optimized instances.

The Drawbacks of Microservices

As Fred Brooks wrote almost 30 years ago, in *The Mythical Man-Month*, there are no silver bullets. Like every other technology, the Microservices architecture pattern has drawbacks. One drawback is the name itself. The term *microservice* places excessive emphasis on service size. In fact, there are some developers who advocate for building extremely fine-grained 10-100 LOC services. While small services are preferable, it's important to remember that small services are a means to an end, and not the primary goal. The goal of microservices is to sufficiently decompose the application in order to facilitate agile application development and deployment.

Another major drawback of microservices is the complexity that arises from the fact that a microservices application is a distributed system. Developers need to choose and implement an inter-process communication mechanism based on either messaging or RPC. Moreover, they must also write code to handle partial failure, since the destination of a request might be slow or unavailable. While none of this is rocket science, it's much more complex than in a monolithic application, where modules invoke one another via language-level method/procedure calls.

Another challenge with microservices is the partitioned database architecture. Business transactions that update multiple business entities are fairly common. These kinds of transactions are trivial to implement in a monolithic application because there is a single database. In a microservices-based application, however, you need to update multiple

databases owned by different services. Using distributed transactions is usually not an option, and not only because of the CAP theorem. They simply are not supported by many of today's highly scalable NoSQL databases and messaging brokers. You end up having to use an eventual consistency-based approach, which is more challenging for developers.

Testing a microservices application is also much more complex. For example, with a modern framework such as Spring Boot, it is trivial to write a test class that starts up a monolithic web application and tests its REST API. In contrast, a similar test class for a service would need to launch that service and any services that it depends upon, or at least configure stubs for those services. Once again, this is not rocket science, but it's important to not underestimate the complexity of doing this.

Another major challenge with the Microservices Architecture pattern is implementing changes that span multiple services. For example, let's imagine that you are implementing a story that requires changes to services A, B, and C, where A depends upon B and B depends upon C. In a monolithic application you could simply change the corresponding modules, integrate the changes, and deploy them in one go. In contrast, in a Microservices Architecture pattern you need to carefully plan and coordinate the rollout of changes to each of the services. For example, you would need to update service C, followed by service B, and then finally service A. Fortunately, most changes typically impact only one service; multi-service changes that require coordination are relatively rare.

Deploying a microservices-based application is also much more complex. A monolithic application is simply deployed on a set of identical servers behind a traditional load balancer. Each application instance is configured with the locations (host and ports) of infrastructure services such as the database and a message broker. In contrast, a microservice application typically consists of a large number of services. For example, Hailo has 160 different services and Netflix has more than 600, according to Adrian Cockcroft.

Each service will have multiple runtime instances. That's many more moving parts that need to be configured, deployed, scaled, and monitored. In addition, you will also need to implement a service discovery mechanism that enables a service to discover the locations (hosts and ports) of any other services it needs to communicate with. Traditional trouble ticket-based and manual approaches to operations cannot scale to this level of complexity. Consequently, successfully deploying a microservices application requires greater control of deployment methods by developers and a high level of automation.

One approach to automation is to use an off-the-shelf platform-as-a-service (PaaS) such as Cloud Foundry. A PaaS provides developers with an easy way to deploy and manage their microservices. It insulates them from concerns such as procuring and configuring IT resources. At the same time, the systems and network professionals who configure the PaaS can ensure compliance with best practices and with company policies.

Another way to automate the deployment of microservices is to develop what is essentially your own PaaS. One typical starting point is to use a clustering solution, such as Kubernetes, in conjunction with a container technology such as Docker. Later in this

ebook we will look at how software-based application delivery approaches like NGINX, which easily handles caching, access control, API metering, and monitoring at the microservice level, can help solve this problem.

Summary

Building complex applications is inherently difficult. The Monolithic Architecture pattern only makes sense for simple, lightweight applications. You will end up in a world of pain if you use it for complex applications. The Microservices Architecture pattern is the better choice for complex, evolving applications, despite the drawbacks and implementation challenges.

In later chapters, I'll dive into the details of various aspects of the Microservices Architecture pattern and discuss topics such as service discovery, service deployment options, and strategies for refactoring a monolithic application into services.

Microservices in Action: NGINX Plus as a Reverse Proxy Server

by Floyd Smith

NGINX powers more than 50% of the top 10,000 websites, and that's largely because of its capabilities as a reverse proxy server. You can "drop NGINX in front of" current applications and even database servers to gain all sorts of capabilities – higher performance, greater security, scalability, flexibility, and more. All with little or no change to your existing application and configuration code. For sites suffering performance stress – or anticipating high loads in the future – the effect may seem little short of miraculous.

So what does this have to do with microservices? Implementing a reverse proxy server, and using the other capabilities of NGINX, gives you architectural flexibility. A reverse proxy server, static and application file caching, and SSL/TLS and HTTP/2 termination all take load off your application, freeing it to "do what only it" – the application – "can do".

NGINX also serves as a load balancer, a crucial role in microservices implementations. The advanced features in NGINX Plus, including sophisticated load-balancing algorithms, multiple methods for session persistence, and management and monitoring, are especially useful with microservices. (NGINX has recently added support for service discovery using DNS `SRV` records, a cutting-edge feature.) And, as mentioned in this chapter, NGINX can help in automating the deployment of microservices.

In addition, NGINX provides the necessary functionality to power the three models in the NGINX Microservices Reference Architecture. The Proxy Model uses NGINX as an API Gateway; the Router Mesh model uses an additional NGINX server as a hub for inter-process communication; and the Fabric Model uses one NGINX server per microservice, controlling HTTP traffic and, optionally, implementing SSL/TLS between microservices, a breakthrough capability.

2 Using an API Gateway

The first chapter in this seven-chapter book about designing, building, and deploying microservices introduced the Microservices Architecture pattern. It discussed the benefits and drawbacks of using microservices and how, despite the complexity of microservices, they are usually the ideal choice for complex applications. This is the second article in the series and will discuss building microservices using an API Gateway.

When you choose to build your application as a set of microservices, you need to decide how your application's clients will interact with the microservices. With a monolithic application there is just one set of endpoints, which are typically replicated, with load balancing used to distribute traffic among them.

In a microservices architecture, however, each microservice exposes a set of what are typically fine-grained endpoints. In this article, we examine how this impacts client-to-application communication and propose an approach that uses an API Gateway.

Introduction

Let's imagine that you are developing a native mobile client for a shopping application. It's likely that you need to implement a product details page, which displays information about any given product.

For example, Figure 2-1 shows what you will see when scrolling through the product details in Amazon's Android mobile application.

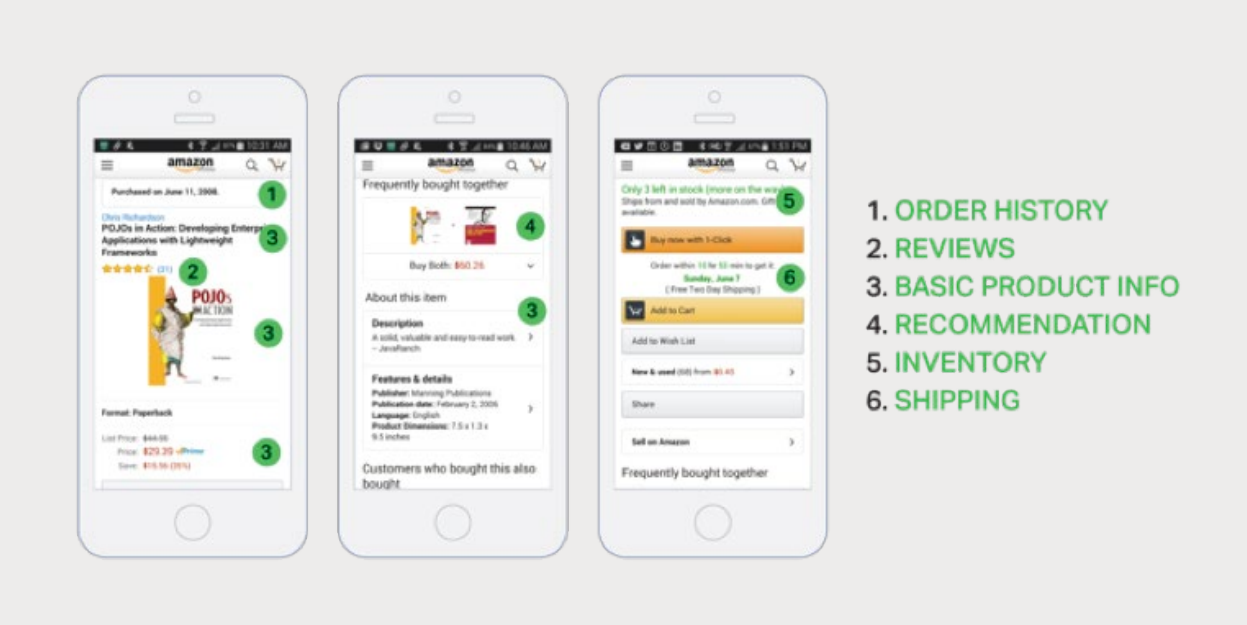


Figure 2-1. A sample shopping application.

Even though this is a smartphone application, the product details page displays a lot of information. For example, not only is there basic product information, such as name, description, and price, but this page also shows:

1. Number of items in the shopping cart
2. Order history
3. Customer reviews
4. Low inventory warning
5. Shipping options
6. Various recommendations, including other products this product is frequently bought with, other products bought by customers who bought this product, and other products viewed by customers who bought this product
7. Alternative purchasing options

When using a monolithic application architecture, a mobile client retrieves this data by making a single REST call to the application, such as:

```
GET api.company.com/productdetails/productId
```

A load balancer routes the request to one of several identical application instances. The application then queries various database tables and return the response to the client.

In contrast, when using the microservices architecture, the data displayed on the product details page is owned by multiple microservices. Here are some of the potential microservices that own data displayed on the sample product-specific page:

- Shopping Cart Service – Number of items in the shopping cart
- Order Service – Order history
- Catalog Service – Basic product information, such as product name, image, and price
- Review Service – Customer reviews
- Inventory Service – Low inventory warning
- Shipping Service – Shipping options, deadlines, and costs, drawn separately from the shipping provider's API
- Recommendation Service(s) – Suggested items

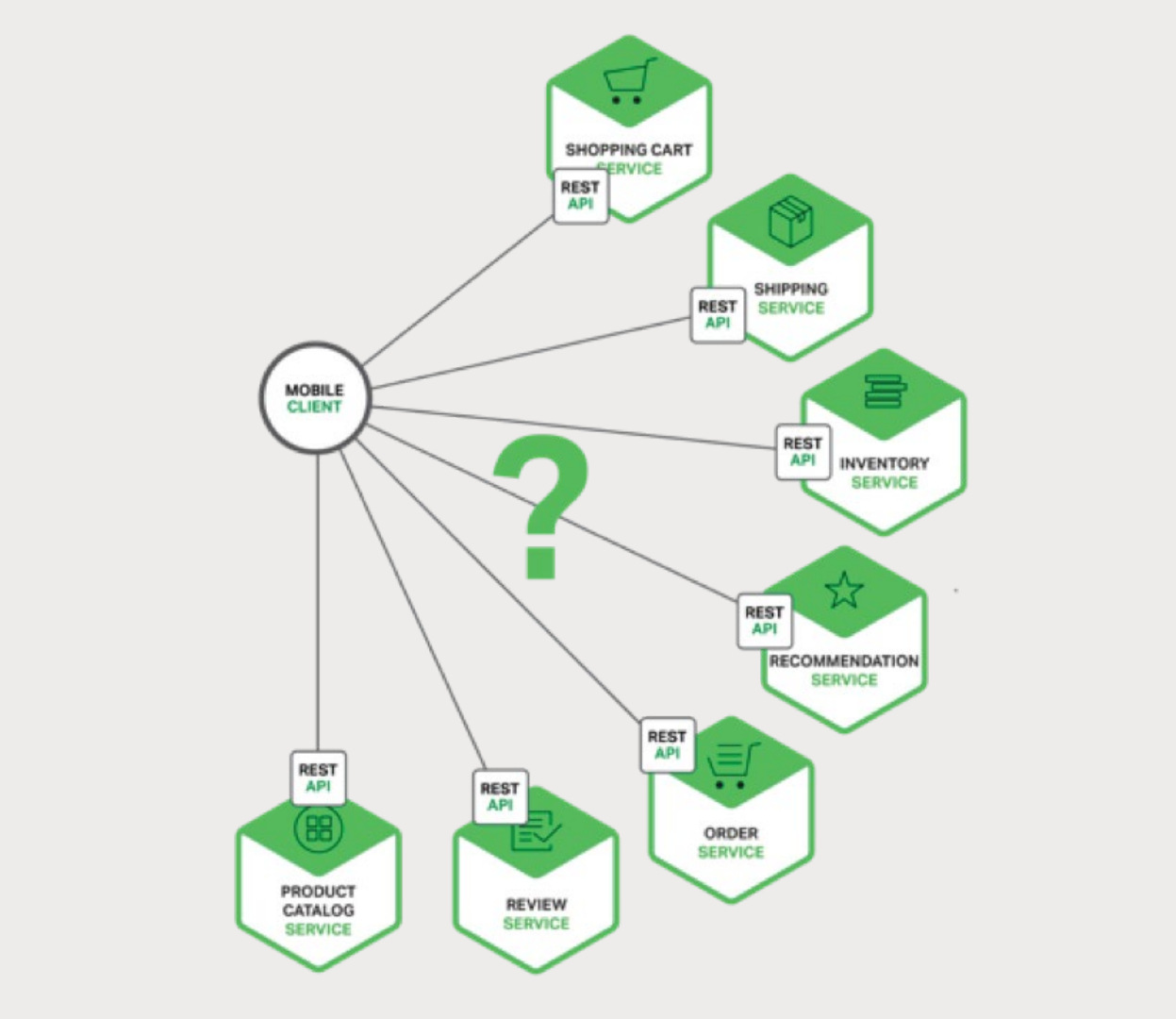


Figure 2-2. Mapping a mobile client's needs to relevant microservices.

We need to decide how the mobile client accesses these services. Let's look at the options.

Direct Client-to-Microservice Communication

In theory, a client could make requests to each of the microservices directly. Each microservice would have a public endpoint:

https://*serviceName*.api.company.name

This URL would map to the microservice's load balancer, which distributes requests across the available instances. To retrieve the product-specific page information, the mobile client would make requests to each of the services listed above.

Unfortunately, there are challenges and limitations with this option. One problem is the mismatch between the needs of the client and the fine-grained APIs exposed by each of the microservices. The client in this example has to make seven separate requests. In more complex applications it might have to make many more. For example, Amazon describes how hundreds of services are involved in rendering their product page. While a client could make that many requests over a LAN, it would probably be too inefficient over the public Internet and would definitely be impractical over a mobile network. This approach also makes the client code much more complex.

Another problem with the client directly calling the microservices is that some might use protocols that are not web-friendly. One service might use Thrift binary RPC while another service might use the AMQP messaging protocol. Neither protocol is particularly browser- or firewall-friendly, and is best used internally. An application should use protocols such as HTTP and WebSocket outside of the firewall.

Another drawback with this approach is that it makes it difficult to refactor the microservices. Over time we might want to change how the system is partitioned into services. For example, we might merge two services or split a service into two or more services. If, however, clients communicate directly with the services, then performing this kind of refactoring can be extremely difficult.

Because of these kinds of problems it rarely makes sense for clients to talk directly to microservices.

Using an API Gateway

Usually a much better approach is to use what is known as an API Gateway. An API Gateway is a server that is the single entry point into the system. It is similar to the Facade pattern from object-oriented design. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.

Figure 2-3 shows how an API Gateway typically fits into the architecture.

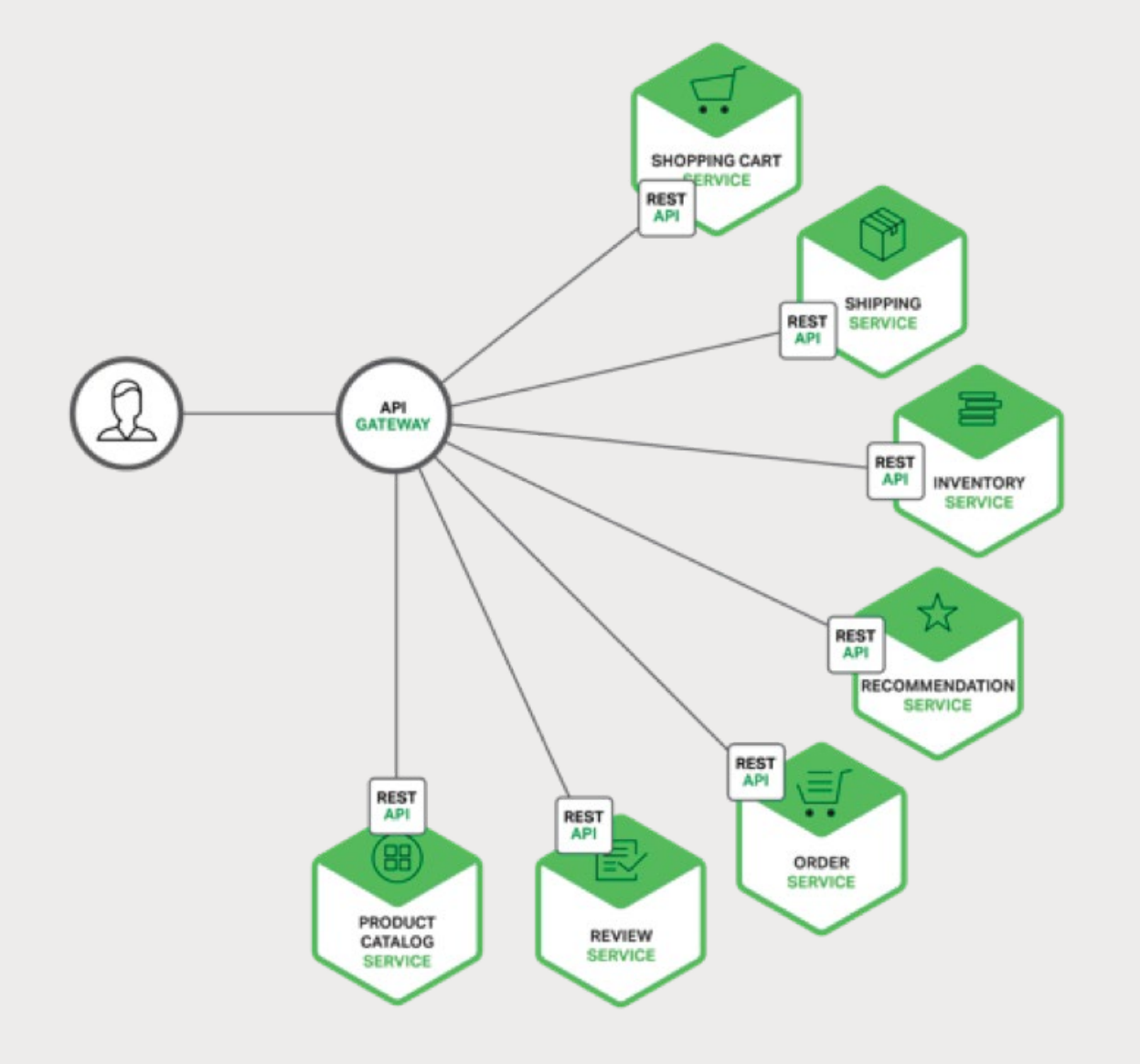


Figure 2-3. Using an API Gateway with microservices.

The API Gateway is responsible for request routing, composition, and protocol translation. All requests from clients first go through the API Gateway. It then routes requests to the appropriate microservice. The API Gateway will often handle a request by invoking multiple microservices and aggregating the results. It can translate between web protocols such as HTTP and WebSocket and web-unfriendly protocols that are used internally.

The API Gateway can also provide each client with a custom API. It typically exposes a coarse-grained API for mobile clients. Consider, for example, the product details scenario. The API Gateway can provide an endpoint **(/productdetails?productid=*xxx*)** that

enables a mobile client to retrieve all of the product details with a single request. The API Gateway handles the request by invoking the various services – product information, recommendations, reviews, etc. – and combining the results.

A great example of an API Gateway is the Netflix API Gateway. The Netflix streaming service is available on hundreds of different kinds of devices including televisions, set-top boxes, smartphones, gaming systems, tablets, etc. Initially, Netflix attempted to provide a one-size-fits-all API for their streaming service. However, they discovered that it didn't work well because of the diverse range of devices and their unique needs. Today, they use an API Gateway that provides an API tailored for each device by running device-specific adapter code. An adapter typically handles each request by invoking, on average, six to seven backend services. The Netflix API Gateway handles billions of requests per day.

Benefits and Drawbacks of an API Gateway

As you might expect, using an API Gateway has both benefits and drawbacks. A major benefit of using an API Gateway is that it encapsulates the internal structure of the application. Rather than having to invoke specific services, clients simply talk to the gateway. The API Gateway provides each kind of client with a specific API. This reduces the number of round trips between the client and application. It also simplifies the client code.

The API Gateway also has some drawbacks. It is yet another highly available component that must be developed, deployed, and managed. There is also a risk that the API Gateway becomes a development bottleneck. Developers must update the API Gateway in order to expose each microservice's endpoints.

It is important that the process for updating the API Gateway be as lightweight as possible. Otherwise, developers will be forced to wait in line in order to update the gateway. Despite these drawbacks, however, for most real-world applications it makes sense to use an API Gateway.

Implementing an API Gateway

Now that we have looked at the motivations and the trade-offs for using an API Gateway, let's look at various design issues you need to consider.

Performance and Scalability

Only a handful of companies operate at the scale of Netflix and need to handle billions of requests per day. However, for most applications the performance and scalability of the API Gateway is usually very important. It makes sense, therefore, to build the API Gateway on a platform that supports asynchronous, non-blocking I/O. There are a variety of different technologies that can be used to implement a scalable API Gateway. On the JVM you can use one of the NIO-based frameworks such Netty, Vertx, Spring Reactor,

or JBoss Undertow. One popular non-JVM option is Node.js, which is a platform built on Chrome's JavaScript engine. Another option is to use NGINX Plus.

NGINX Plus offers a mature, scalable, high-performance web server and reverse proxy that is easily deployed, configured, and programmed. NGINX Plus can manage authentication, access control, load balancing requests, caching responses, and provides application-aware health checks and monitoring.

Using a Reactive Programming Model

The API Gateway handles some requests by simply routing them to the appropriate backend service. It handles other requests by invoking multiple backend services and aggregating the results. With some requests, such as a product details request, the requests to backend services are independent of one another. In order to minimize response time, the API Gateway should perform independent requests concurrently.

Sometimes, however, there are dependencies between requests. The API Gateway might first need to validate the request by calling an authentication service before routing the request to a backend service. Similarly, to fetch information about the products in a customer's wish list, the API Gateway must first retrieve the customer's profile containing that information, and then retrieve the information for each product. Another interesting example of API composition is the Netflix Video Grid.

Writing API composition code using the traditional asynchronous callback approach quickly leads you to callback hell. The code will be tangled, difficult to understand, and error-prone. A much better approach is to write API Gateway code in a declarative style using a reactive approach. Examples of reactive abstractions include Future in Scala, CompletableFuture in Java 8, and Promise in JavaScript. There is also Reactive Extensions (also called Rx or ReactiveX), which was originally developed by Microsoft for the .NET platform. Netflix created RxJava for the JVM specifically to use in their API Gateway. There is also RxJS for JavaScript, which runs in both the browser and Node.js. Using a reactive approach will enable you to write simple yet efficient API Gateway code.

Service Invocation

A microservices-based application is a distributed system and must use an inter-process communication mechanism. There are two styles of inter-process communication. One option is to use an asynchronous, messaging-based mechanism. Some implementations use a message broker such as JMS or AMQP. Others, such as Zeromq, are brokerless and the services communicate directly.

The other style of inter-process communication is a synchronous mechanism such as HTTP or Thrift. A system will typically use both asynchronous and synchronous styles. It might even use multiple implementations of each style. Consequently, the API Gateway will need to support a variety of communication mechanisms.

Service Discovery

The API Gateway needs to know the location (IP address and port) of each microservice with which it communicates. In a traditional application, you could probably hardwire the locations, but in a modern, cloud-based microservices application, finding the needed locations is a non-trivial problem.

Infrastructure services, such as a message broker, will usually have a static location, which can be specified via OS environment variables. However, determining the location of an application service is not so easy.

Application services have dynamically assigned locations. Also, the set of instances of a service changes dynamically because of autoscaling and upgrades. Consequently, the API Gateway, like any other service client in the system, needs to use the system's service discovery mechanism: either server-side discovery or client-side discovery. Chapter 4 describes service discovery in more detail. For now, it is worthwhile to note that if the system uses client-side discovery, then the API Gateway must be able to query the service registry, which is a database of all microservice instances and their locations.

Handling Partial Failures

Another issue you have to address when implementing an API Gateway is the problem of partial failure. This issue arises in all distributed systems whenever one service calls another service that is either responding slowly or is unavailable. The API Gateway should never block indefinitely waiting for a downstream service. However, how it handles the failure depends on the specific scenario and which service is failing. For example, if the recommendation service is unresponsive in the product details scenario, the API Gateway should return the rest of the product details to the client since they are still useful to the user. The recommendations could either be empty or replaced by, for example, a hardwired top ten list. If, however, the product information service is unresponsive, then the API Gateway should return an error to the client.

The API Gateway could also return cached data if that is available. For example, since product prices change infrequently, the API Gateway could return cached pricing data if the pricing service is unavailable. The data can be cached by the API Gateway itself or be stored in an external cache, such as Redis or Memcached. By returning either default data or cached data, the API Gateway ensures that system failures minimally impact the user experience.

Netflix Hystrix is an incredibly useful library for writing code that invokes remote services. Hystrix times out calls that exceed the specified threshold. It implements a *circuit breaker* pattern, which stops the client from waiting needlessly for an unresponsive service. If the error rate for a service exceeds a specified threshold, Hystrix trips the circuit breaker and all requests will fail immediately for a specified period of time. Hystrix lets you define a fallback action when a request fails, such as reading from a cache or returning a default value. If you are using the JVM you should definitely consider using Hystrix. And, if you are running in a non-JVM environment, you should use an equivalent library.

Summary

For most microservices-based applications, it makes sense to implement an API Gateway, which acts as a single entry point into a system. The API Gateway is responsible for request routing, composition, and protocol translation. It provides each of the application's clients with a custom API. The API Gateway can also mask failures in the backend services by returning cached or default data. In the next chapter, we will look at communication between services.

Microservices in Action: NGINX Plus as an API Gateway

by Floyd Smith

This chapter discusses how an API Gateway serves as a single entry point into a system. And it can handle other functions such as load balancing, caching, monitoring, protocol translation, and others – while NGINX, when implemented as a reverse proxy server, functions as a single entry point into a system and supports all the additional functions mentioned for an API Gateway. So using NGINX as the host for an API Gateway can work very well indeed.

Thinking of NGINX as an API Gateway is not an idea that's original to this ebook. NGINX Plus is a leading platform for managing and securing HTTP-based API traffic. You can implement your own API Gateway or use an existing API management platform, many of which leverage NGINX.

Reasons for using NGINX Plus as an API Gateway include:

- **Access management** – You can use a variety of access control list (ACL) methods and easily implement SSL/TLS, either at the web application level as is typical, or also down to the level of each individual microservice.
- **Manageability and resilience** – You can update your NGINX Plus-based API server without downtime, using the NGINX dynamic reconfiguration API, a Lua module, Perl, live restarts without downtime, or changes driven by Chef, Puppet, ZooKeeper, or DNS.
- **Integration with third-party tools** – NGINX Plus is already integrated with leading-edge tools such as 3scale, Kong, and the MuleSoft integration platform (to mention only tools described on the NGINX website.)

NGINX Plus is used extensively as an API Gateway in the NGINX Microservices Reference Architecture. Use the articles assembled here and, when publicly available, the MRA, for examples of how to implement this in your own applications.

3 Inter-Process Communication

This is the third chapter in this ebook about building applications with a microservices architecture. Chapter 1 introduces the Microservices Architecture pattern, compares it with the Monolithic Architecture pattern, and discusses the benefits and drawbacks of using microservices. Chapter 2 describes how clients of an application communicate with the microservices via an intermediary known as an API Gateway. In this chapter, we take a look at how the services within a system communicate with one another. Chapter 4 explores the closely related problem of service discovery.

Introduction

In a monolithic application, components invoke one another via language-level method or function calls. In contrast, a microservices-based application is a distributed system running on multiple machines. Each service instance is typically a process.

Consequently, as Figure 3-1 shows, services must interact using an inter-process communication (IPC) mechanism.

Later on we will look at specific IPC technologies, but first let's explore various design issues.

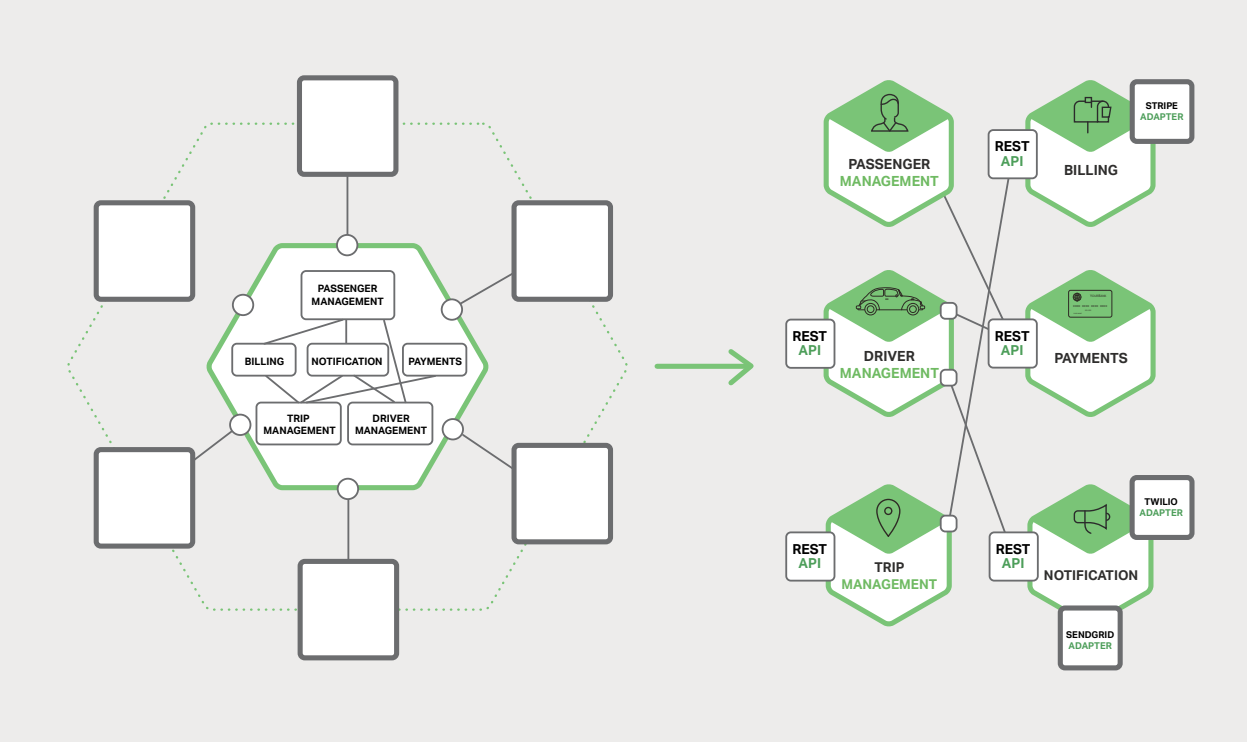


Figure 3-1. Microservices use inter-process communication to interact.

Interaction Styles

When selecting an IPC mechanism for a service, it is useful to think first about how services interact. There are a variety of client↔service interaction styles. They can be categorized along two dimensions. The first dimension is whether the interaction is one-to-one or one-to-many:

- One-to-one – Each client request is processed by exactly one service instance.
- One-to-many – Each request is processed by multiple service instances.

The second dimension is whether the interaction is synchronous or asynchronous:

- Synchronous – The client expects a timely response from the service and might even block while it waits.
- Asynchronous – The client doesn't block while waiting for a response, and the response, if any, isn't necessarily sent immediately.

The following table shows the various interaction styles.

	ONE-TO-ONE	ONE-TO-MANY
SYNCHRONOUS	Request/response	—
ASYNCHRONOUS	Notification	Publish/subscribe
	Request/async response	Publish/async responses

Table 3-1. Inter-process communication styles.

There are the following kinds of one-to-one interactions, both synchronous (request/response) and asynchronous (notification and request/async response):

- Request/response – A client makes a request to a service and waits for a response. The client expects the response to arrive in a timely fashion. In a thread-based application, the thread that makes the request might even block while waiting.
- Notification (a.k.a. a one-way request) – A client sends a request to a service but no reply is expected or sent.
- Request/async response – A client sends a request to a service, which replies asynchronously. The client does not block while waiting and is designed with the assumption that the response might not arrive for a while.

There are the following kinds of one-to-many interactions, both of which are asynchronous:

- Publish/subscribe – A client publishes a notification message, which is consumed by zero or more interested services.
- Publish/async responses – A client publishes a request message, and then waits a certain amount of time for responses from interested services.

Each service typically uses a combination of these interaction styles. For some services, a single IPC mechanism is sufficient. Other services might need to use a combination of IPC mechanisms.

Figure 3-2 shows how services in a taxi-hailing application might interact when the user requests a trip.

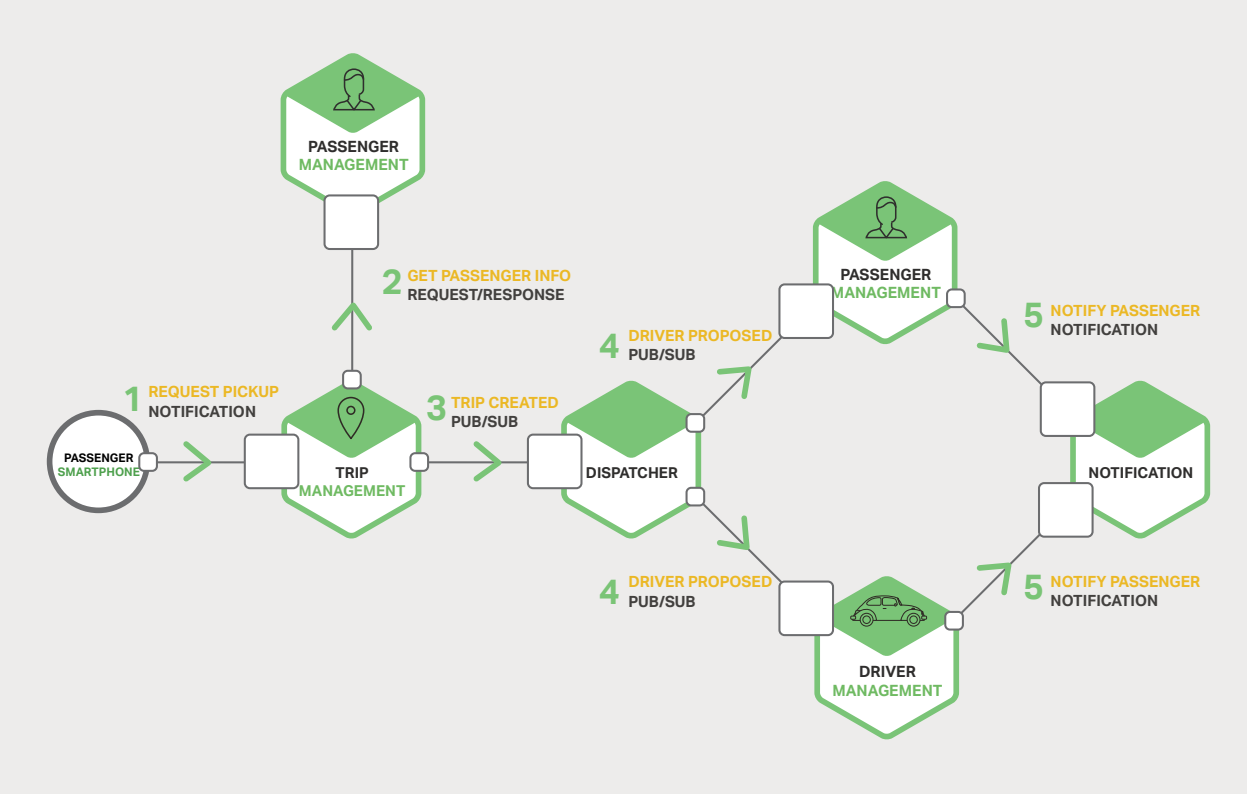


Figure 3-2. Using multiple IPC mechanisms for service interactions.

The services use a combination of notifications, request/response, and publish/subscribe. For example, the passenger's smartphone sends a notification to the Trip Management service to request a pickup. The Trip Management service verifies that the passenger's account is active by using request/response to invoke the Passenger Management service. The Trip Management service then creates the trip and uses publish/subscribe to notify other services including the Dispatcher, which locates an available driver.

Now that we have looked at interaction styles, let's take a look at how to define APIs.

Defining APIs

A service's API is a contract between the service and its clients. Regardless of your choice of IPC mechanism, it's important to precisely define a service's API using some kind of interface definition language (IDL). There are even good arguments for using an API-first approach to defining services. You begin the development of a service by writing the interface definition and reviewing it with the client developers. It is only after iterating on the API definition that you implement the service. Doing this design up front increases your chances of building a service that meets the needs of its clients.

As you will see later in this article, the nature of the API definition depends on which IPC mechanism you are using. If you are using messaging, the API consists of the message channels and the message types. If you are using HTTP, the API consists of the URLs and the request and response formats. Later on we will describe some IDLs in more detail.

Evolving APIs

A service's API invariably changes over time. In a monolithic application it is usually straightforward to change the API and update all the callers. In a microservices-based application it is a lot more difficult, even if all of the consumers of your API are other services in the same application. You usually cannot force all clients to upgrade in lockstep with the service. Also, you will probably incrementally deploy new versions of a service such that both old and new versions of a service will be running simultaneously. It is important to have a strategy for dealing with these issues.

How you handle an API change depends on the size of the change. Some changes are minor and backward compatible with the previous version. You might, for example, add attributes to requests or responses. It makes sense to design clients and services so that they observe the robustness principle. Clients that use an older API should continue to work with the new version of the service. The service provides default values for the missing request attributes and the clients ignore any extra response attributes. It is important to use an IPC mechanism and a messaging format that enable you to easily evolve your APIs.

Sometimes, however, you must make major, incompatible changes to an API. Since you can't force clients to upgrade immediately, a service must support older versions of the

API for some period of time. If you are using an HTTP-based mechanism such as REST, one approach is to embed the version number in the URL. Each service instance might handle multiple versions simultaneously. Alternatively, you could deploy different instances that each handle a particular version.

Handling Partial Failure

As mentioned in Chapter 2 about the API Gateway, in a distributed system there is the ever-present risk of partial failure. Since clients and services are separate processes, a service might not be able to respond in a timely way to a client's request. A service might be down because of a failure or for maintenance. Or the service might be overloaded and responding extremely slowly to requests.

Consider, for example, the product details scenario from Chapter 2. Let's imagine that the Recommendation Service is unresponsive. A naive implementation of a client might block indefinitely waiting for a response. Not only would that result in a poor user experience, but also, in many applications it would consume a precious resource such as a thread. Eventually the runtime would run out of threads and become unresponsive, as shown in Figure 3-3.

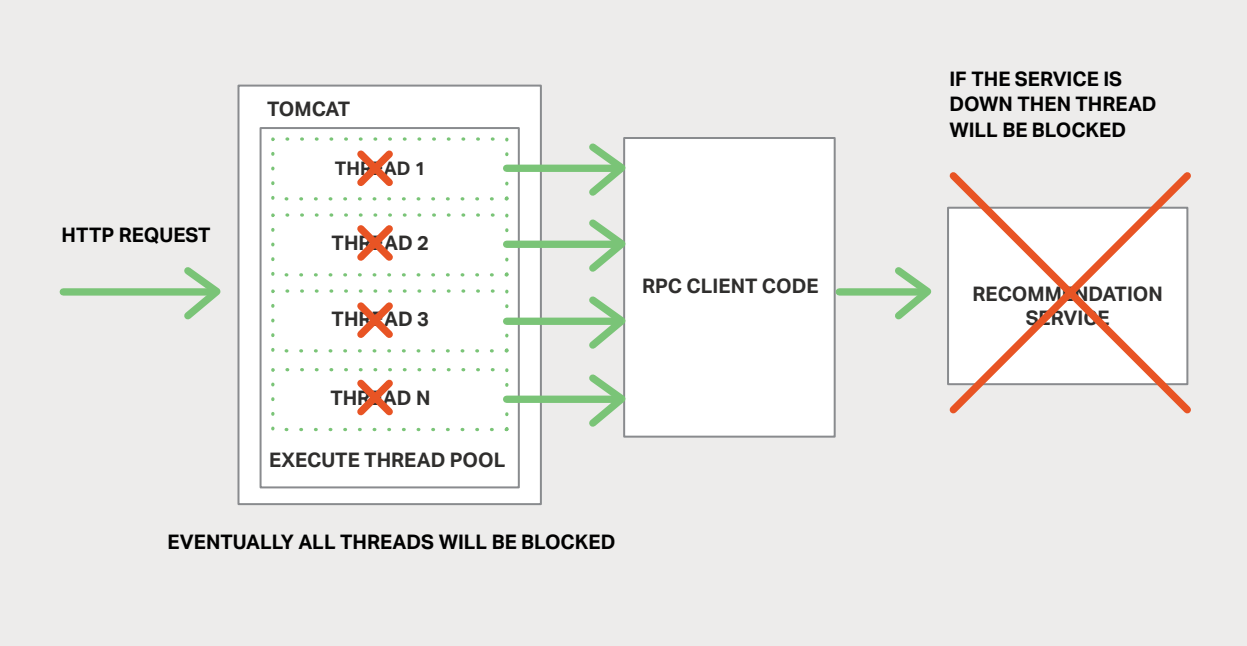


Figure 3-3. Threads block due to an unresponsive service.

To prevent this problem, it is essential that you design your services to handle partial failures.

A good approach to follow is the one described by Netflix. The strategies for dealing with partial failures include:

- Network timeouts – Never block indefinitely and always use timeouts when waiting for a response. Using timeouts ensures that resources are never tied up indefinitely.
- Limiting the number of outstanding requests – Impose an upper bound on the number of outstanding requests that a client can have with a particular service. If the limit has been reached, it is probably pointless to make additional requests, and those attempts need to fail immediately.
- Circuit breaker pattern – Track the number of successful and failed requests. If the error rate exceeds a configured threshold, trip the circuit breaker so that further attempts fail immediately. If a large number of requests are failing, that suggests the service is unavailable and that sending requests is pointless. After a timeout period, the client should try again and, if successful, close the circuit breaker.
- Provide fallbacks – Perform fallback logic when a request fails. For example, return cached data or a default value, such as an empty set of recommendations.

Netflix Hystrix is an open source library that implements these and other patterns. If you are using the JVM you should definitely consider using Hystrix. And, if you are running in a non-JVM environment, you should use an equivalent library.

IPC Technologies

There are lots of different IPC technologies to choose from. Services can use synchronous request/response-based communication mechanisms such as HTTP-based REST or Thrift. Alternatively, they can use asynchronous, message-based communication mechanisms such as AMQP or STOMP.

There are also a variety of different message formats. Services can use human readable, text-based formats such as JSON or XML. Alternatively, they can use a binary format (which is more efficient) such as Avro or Protocol Buffers. Later on we will look at synchronous IPC mechanisms, but first let's discuss asynchronous IPC mechanisms.

Asynchronous, Message-Based Communication

When using messaging, processes communicate by asynchronously exchanging messages. A client makes a request to a service by sending it a message. If the service is expected to reply, it does so by sending a separate message back to the client. Since the communication is asynchronous, the client does not block waiting for a reply. Instead, the client is written assuming that the reply will not be received immediately.

A message consists of headers (metadata such as the sender) and a message body. Messages are exchanged over channels. Any number of producers can send messages to a channel. Similarly, any number of consumers can receive messages from a channel. There are two kinds of channels, point-to-point and publish-subscribe:

- A point-to-point channel delivers a message to exactly one of the consumers that are reading from the channel. Services use point-to-point channels for the one-to-one interaction styles described earlier.
- A publish-subscribe channel delivers each message to all of the attached consumers. Services use publish-subscribe channels for the one-to-many interaction styles described above.

Figure 3-4 shows how the taxi-hailing application might use publish-subscribe channels.

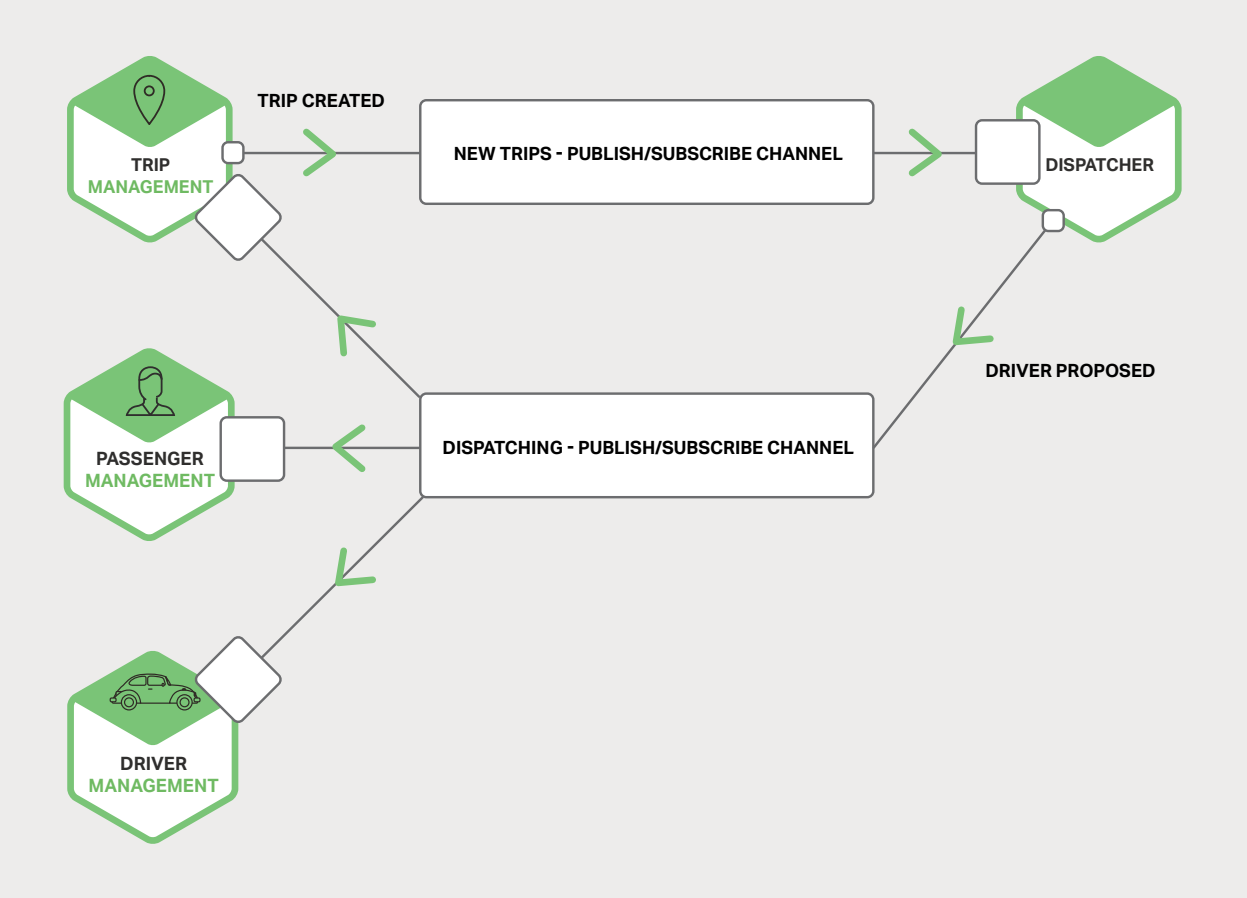


Figure 3-4. Using publish-subscribe channels in a taxi-hailing application.

The Trip Management service notifies interested services, such as the Dispatcher, about a new Trip by writing a Trip Created message to a publish-subscribe channel. The Dispatcher finds an available driver and notifies other services by writing a Driver Proposed message to a publish-subscribe channel.

There are many messaging systems to choose from. You should pick one that supports a variety of programming languages.

Some messaging systems support standard protocols such as AMQP and STOMP. Other messaging systems have proprietary but documented protocols.

There are a large number of open source messaging systems to choose from, including RabbitMQ, Apache Kafka, Apache ActiveMQ, and NSQ. At a high level, they all support some form of messages and channels. They all strive to be reliable, high-performance, and scalable. However, there are significant differences in the details of each broker's messaging model.

There are many advantages to using messaging:

- Decouples the client from the service – A client makes a request simply by sending a message to the appropriate channel. The client is completely unaware of the service instances. It does not need to use a discovery mechanism to determine the location of a service instance.
- Message buffering – With a synchronous request/response protocol, such as HTTP, both the client and service must be available for the duration of the exchange. In contrast, a message broker queues up the messages written to a channel until the consumer can process them. This means, for example, that an online store can accept orders from customers even when the order fulfillment system is slow or unavailable. The order messages simply queue up.
- Flexible client-service interactions – Messaging supports all of the interaction styles described earlier.
- Explicit inter-process communication – RPC-based mechanisms attempt to make invoking a remote service look the same as calling a local service. However, because of the laws of physics and the possibility of partial failure, they are in fact quite different. Messaging makes these differences very explicit so developers are not lulled into a false sense of security.

There are, however, some downsides to using messaging:

- Additional operational complexity – The messaging system is yet another system component that must be installed, configured, and operated. It's essential that the message broker be highly available, otherwise system reliability is impacted.
- Complexity of implementing request/response-based interaction – Request/response-style interaction requires some work to implement. Each request message must contain a reply channel identifier and a correlation identifier. The service writes a response message containing the correlation ID to the reply channel. The client uses the correlation ID to match the response with the request. It is often easier to use an IPC mechanism that directly supports request/response.

Now that we have looked at using messaging-based IPC, let's examine request/response-based IPC.

Synchronous, Request/Response IPC

When using a synchronous, request/response-based IPC mechanism, a client sends a request to a service. The service processes the request and sends back a response.

In many clients, the thread that makes the request blocks while waiting for a response. Other clients might use asynchronous, event-driven client code that is perhaps encapsulated by Futures or Rx Observables. However, unlike when using messaging, the client assumes that the response will arrive in a timely fashion.

There are numerous protocols to choose from. Two popular protocols are REST and Thrift. Let's first take a look at REST.

REST

Today it is fashionable to develop APIs in the RESTful style. REST is an IPC mechanism that (almost always) uses HTTP.

A key concept in REST is a resource, which typically represents a business object such as a Customer or Product, or a collection of such business objects. REST uses the HTTP verbs for manipulating resources, which are referenced using a URL. For example, a `GET` request returns the representation of a resource, which might be in the form of an XML document or JSON object. A `POST` request creates a new resource, and a `PUT` request updates a resource.

To quote Roy Fielding, the creator of REST:

"REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems."
—Roy Fielding, *Architectural Styles and the Design of Network-based Software Architectures*

Figure 3-5 shows one of the ways that the taxi-hailing application might use REST.

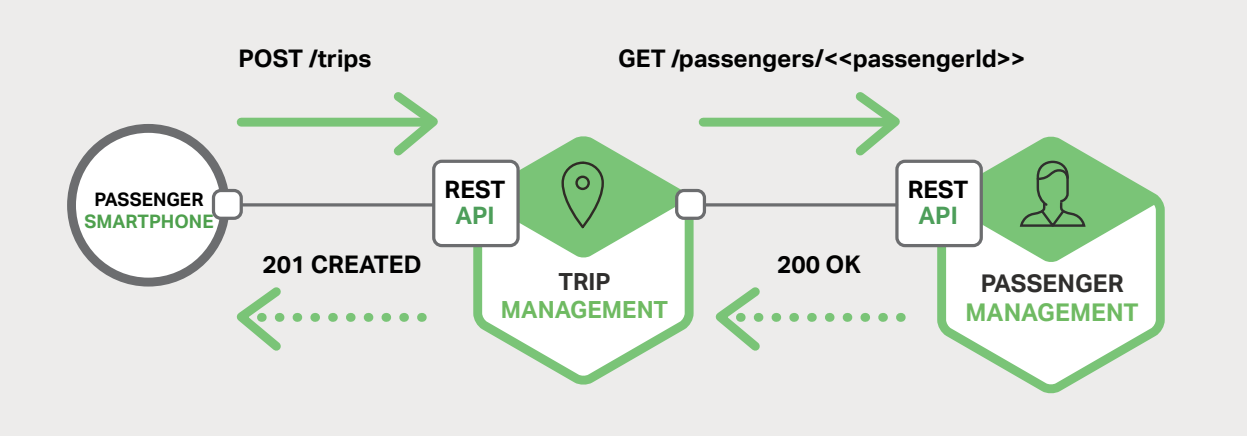


Figure 3-5. A taxi-hailing application uses RESTful interaction.

The passenger's smartphone requests a trip by making a `POST` request to the **/trips** resource of the Trip Management service. This service handles the request by sending a `GET` request for information about the passenger to the Passenger Management service. After verifying that the passenger is authorized to create a trip, the Trip Management service creates the trip and returns a `201` response to the smartphone.

Many developers claim their HTTP-based APIs are RESTful. However, as Fielding describes in this blog post, not all of them actually are.

Leonard Richardson (no relation) defines a very useful maturity model for REST that consists of the following levels:

- Level 0 – Clients of a level 0 API invoke the service by making HTTP `POST` requests to its sole URL endpoint. Each request specifies the action to perform, the target of the action (for example, the business object), and any parameters.
- Level 1 – A level 1 API supports the idea of resources. To perform an action on a resource, a client makes a `POST` request that specifies the action to perform and any parameters.
- Level 2 – A level 2 API uses HTTP verbs to perform actions: `GET` to retrieve, `POST` to create, and `PUT` to update. The request query parameters and body, if any, specify the action's parameters. This enables services to leverage web infrastructure such as caching for `GET` requests.
- Level 3 – The design of a level 3 API is based on the terribly named principle, HATEOAS (Hypertext As The Engine Of Application State). The basic idea is that the representation of a resource returned by a `GET` request contains links for performing the allowable actions on that resource. For example, a client can cancel an order using a link in the Order representation returned in response to the `GET` request sent to retrieve the order.

  One of the benefits of HATEOAS is include no longer having to hardwire URLs into client code. Another benefit is that because the representation of a resource contains links for the allowable actions, the client doesn't have to guess what actions can be performed on a resource in its current state.

There are numerous benefits to using a protocol that is based on HTTP:

- HTTP is simple and familiar.
- You can test an HTTP API from within a browser using an extension such as Postman, or from the command line using `curl` (assuming JSON or some other text format is used).
- It directly supports request/response-style communication.
- HTTP is, of course, firewall-friendly.
- It doesn't require an intermediate broker, which simplifies the system's architecture.

There are some drawbacks to using HTTP:

- HTTP only directly supports the request/response style of interaction. You can use HTTP for notifications but the server must always send an HTTP response.
- Because the client and service communicate directly (without an intermediary to buffer messages), they must both be running for the duration of the exchange.
- The client must know the location (that is, the URL) of each service instance. As described in Chapter 2 about the API Gateway, this is a non-trivial problem in a modern application. Clients must use a service discovery mechanism to locate service instances.

The developer community has recently rediscovered the value of an interface definition language for RESTful APIs. There are a few options, including RAML and Swagger. Some IDLs, such as Swagger, allow you to define the format of request and response messages. Others, such as RAML, require you to use a separate specification such as JSON Schema. As well as describing APIs, IDLs typically have tools that generate client stubs and server skeletons from an interface definition.

Thrift

Apache Thrift is an interesting alternative to REST. It is a framework for writing cross-language RPC clients and servers. Thrift provides a C-style IDL for defining your APIs. You use the Thrift compiler to generate client-side stubs and server-side skeletons. The compiler generates code for a variety of languages including C++, Java, Python, PHP, Ruby, Erlang, and Node.js.

A Thrift interface consists of one or more services. A service definition is analogous to a Java interface. It is a collection of strongly typed methods.

Thrift methods can either return a (possibly void) value or, if they are defined as one-way, no value. Methods that return a value implement the request/response style of interaction; the client waits for a response and might throw an exception. One-way methods correspond to the notification style of interaction; the server does not send a response.

Thrift supports various message formats: JSON, binary, and compact binary. Binary is more efficient than JSON because it is faster to decode. And, as the name suggests, compact binary is a space-efficient format. JSON is, of course, human- and browser-friendly. Thrift also gives you a choice of transport protocols including raw TCP and HTTP. Raw TCP is likely to be more efficient than HTTP. However, HTTP is firewall-friendly, browser-friendly, and human-friendly.

Message Formats

Now that we have looked at HTTP and Thrift, let's examine the issue of message formats. If you are using a messaging system or REST, you get to pick your message format. Other IPC mechanisms such as Thrift might support only a small number of message formats, or even just one. In either case, it's important to use a cross-language message

format. Even if you are writing your microservices in a single language today, it's likely that you will use other languages in the future.

There are two main kinds of message formats: text and binary. Examples of text-based formats include JSON and XML. An advantage of these formats is that not only are they human-readable, they are self-describing. In JSON, the attributes of an object are represented by a collection of name-value pairs. Similarly, in XML the attributes are represented by named elements and values. This enables a consumer of a message to pick out the values that it is interested in and ignore the rest. Consequently, minor changes to the message format can be easily made backward compatible.

The structure of XML documents is specified by an XML schema. Over time, the developer community has come to realize that JSON also needs a similar mechanism. One option is to use JSON Schema, either stand-alone or as part of an IDL such as Swagger.

A downside of using a text-based message format is that the messages tend to be verbose, especially XML. Because the messages are self-describing, every message contains the name of the attributes in addition to their values. Another drawback is the overhead of parsing text. Consequently, you might want to consider using a binary format.

There are several binary formats to choose from. If you are using Thrift RPC, you can use binary Thrift. If you get to pick the message format, popular options include Protocol Buffers and Apache Avro. Both of these formats provide a typed IDL for defining the structure of your messages. One difference, however, is that Protocol Buffers uses tagged fields, whereas an Avro consumer needs to know the schema in order to interpret messages. As a result, API evolution is easier with Protocol Buffers than with Avro. This blog post is an excellent comparison of Thrift, Protocol Buffers, and Avro.

Summary

Microservices must communicate using an inter-process communication mechanism. When designing how your services will communicate, you need to consider various issues: how services interact, how to specify the API for each service, how to evolve the APIs, and how to handle partial failure. There are two kinds of IPC mechanisms that microservices can use: asynchronous messaging and synchronous request/response. In order to communicate, one service must be able to find another. In Chapter 4 we will look at the problem of service discovery in a microservices architecture.

Microservices in Action: NGINX and Application Architecture

by Floyd Smith

NGINX enables you to implement various scaling and mirroring options that make your application more responsive and highly available. The choices you make for scaling and mirroring affect how you do inter-process communication, the topic of this chapter.

We at NGINX recommend that you consider a four-tier architecture when implementing your microservices-based application. Forrester has a detailed report on the topic which you can download, at no charge, from NGINX.

The tiers represent clients (the newest layer – including desktop or laptop and mobile, wearable, or IoT clients), delivery, aggregation (including data storage), and services, which incorporate application functionality and service-specific, rather than shared, data stores.

The four-tier architecture is much more flexible, scalable, responsive, mobile-friendly, and inherently supportive of microservices-based application development and delivery than the previous, three-tier architecture. Industry leaders such as Netflix and Uber are able to achieve the level of performance their users demand because they use this kind of architecture.

NGINX is inherently well-suited to the four-tier architecture, with capabilities ranging from media streaming for the client tier, to load balancing and caching for the delivery tier, tools for high-performance and secure API-based communication at the aggregation tier, and support for flexible management of ephemeral services instances in the services tier.

This same flexibility makes it possible to implement robust scaling and mirroring patterns for handling changes in traffic volumes, to protect against security attacks, and to provide high availability with failover configurations available at a moment's notice.

In these more complex architectures, which include service instance instantiation as demand requires and the need for constant service discovery, decoupled inter-process communications tend to be favored. The asynchronous and one-to-many communication styles here may be more flexible, and ultimately offer higher performance and reliability, than tightly coupled communication styles.

4 Service Discovery

This is the fourth chapter in this ebook, which is about building applications with microservices. Chapter 1 introduces the Microservices Architecture pattern and discussed the benefits and drawbacks of using microservices. Chapter 2 and Chapter 3 describe different aspects of communication between microservices. In this chapter, we explore the closely related problem of service discovery.

Why Use Service Discovery?

Let's imagine that you are writing some code that invokes a service that has a REST API or Thrift API. In order to make a request, your code needs to know the network location (IP address and port) of a service instance. In a traditional application running on physical hardware, the network locations of service instances are relatively static. For example, your code can read the network locations from a configuration file that is occasionally updated.

In a modern, cloud-based microservices application, however, this is a much more difficult problem to solve, as shown in Figure 4-1.

Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, your client code needs to use a more elaborate service discovery mechanism.

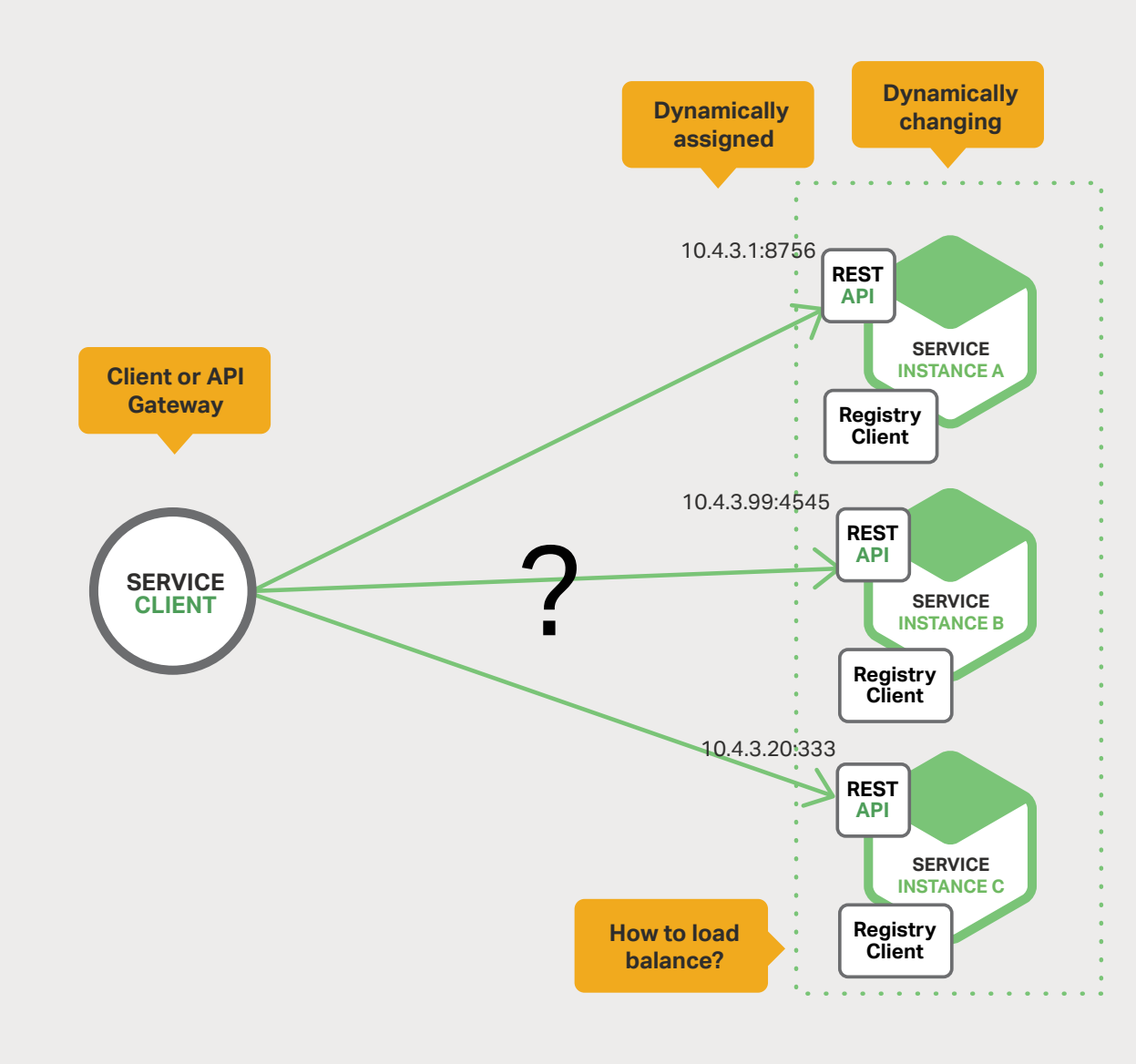


Figure 4-1. A client or API Gateway needs help finding services.

There are two main service discovery patterns: client-side discovery and server-side discovery. Let's first look at client-side discovery.

The Client-Side Discovery Pattern

When using client-side discovery pattern, the client is responsible for determining the network locations of available service instances and load balancing requests across them. The client queries a service registry, which is a database of available service instances. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.

Figure 4-2 shows the structure of this pattern:

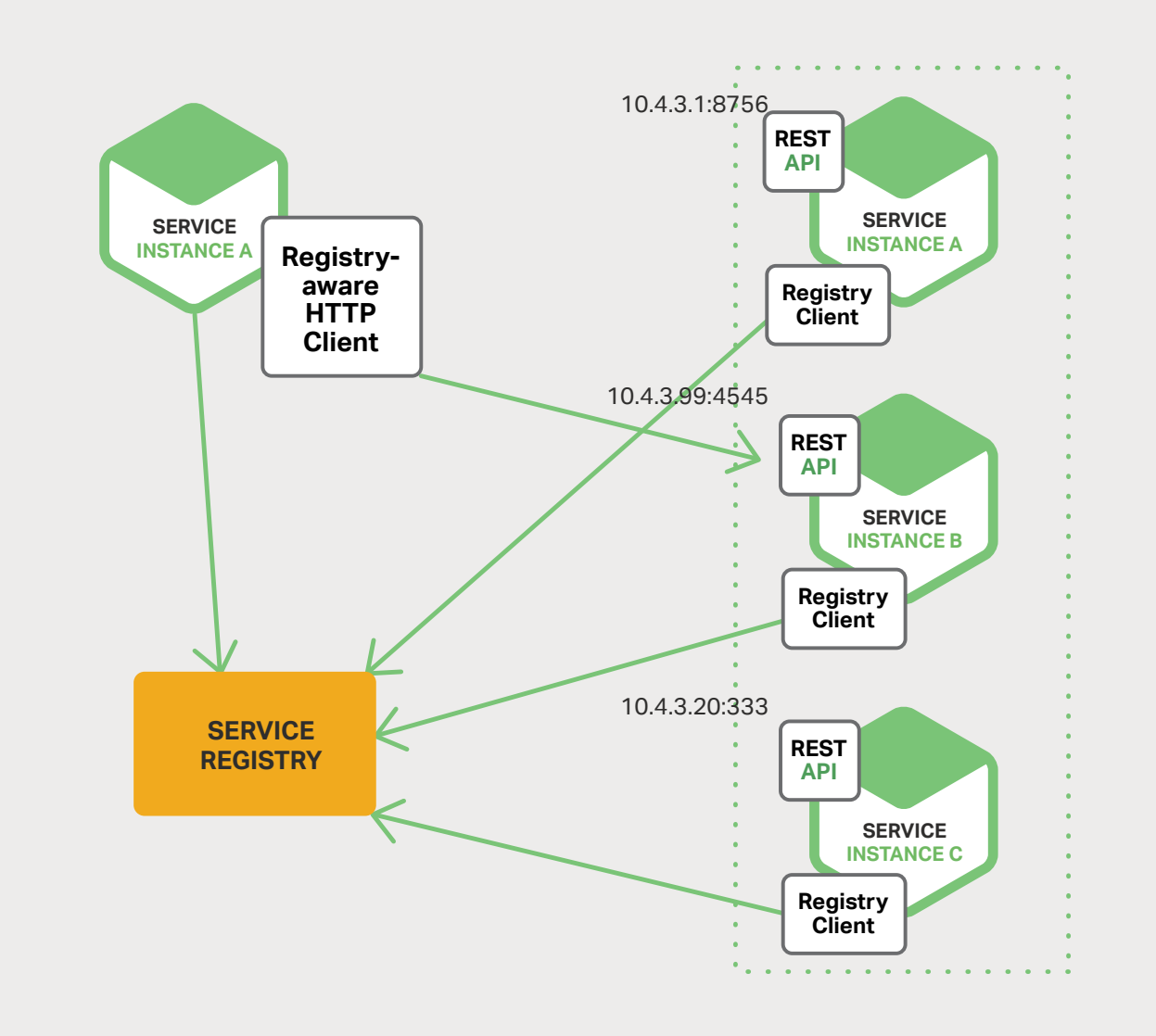


Figure 4-2. Clients can take on the task of discovering services.

The network location of a service instance is registered with the service registry when it starts up. It is removed from the service registry when the instance terminates. The service instance's registration is typically refreshed periodically using a heartbeat mechanism.

Netflix OSS provides a great example of the client-side discovery pattern. Netflix Eureka is a service registry. It provides a REST API for managing service-instance registration and for querying available instances. Netflix Ribbon is an IPC client that works with Eureka to load balance requests across the available service instances. We will discuss Eureka in more depth later in this chapter.

The client-side discovery pattern has a variety of benefits and drawbacks. This pattern is relatively straightforward and, except for the service registry, there are no other moving parts. Also, since the client knows about the available services instances, it can make intelligent, application-specific load-balancing decisions such as using hashing consistently. One significant drawback of this pattern is that it couples the client with the service registry. You must implement client-side service discovery logic for each programming language and framework used by your service clients.

Now that we have looked at client-side discovery, let's take a look at server-side discovery.

The Server-Side Discovery Pattern

The other approach to service discovery is the server-side discovery pattern. Figure 4-3 shows the structure of this pattern:

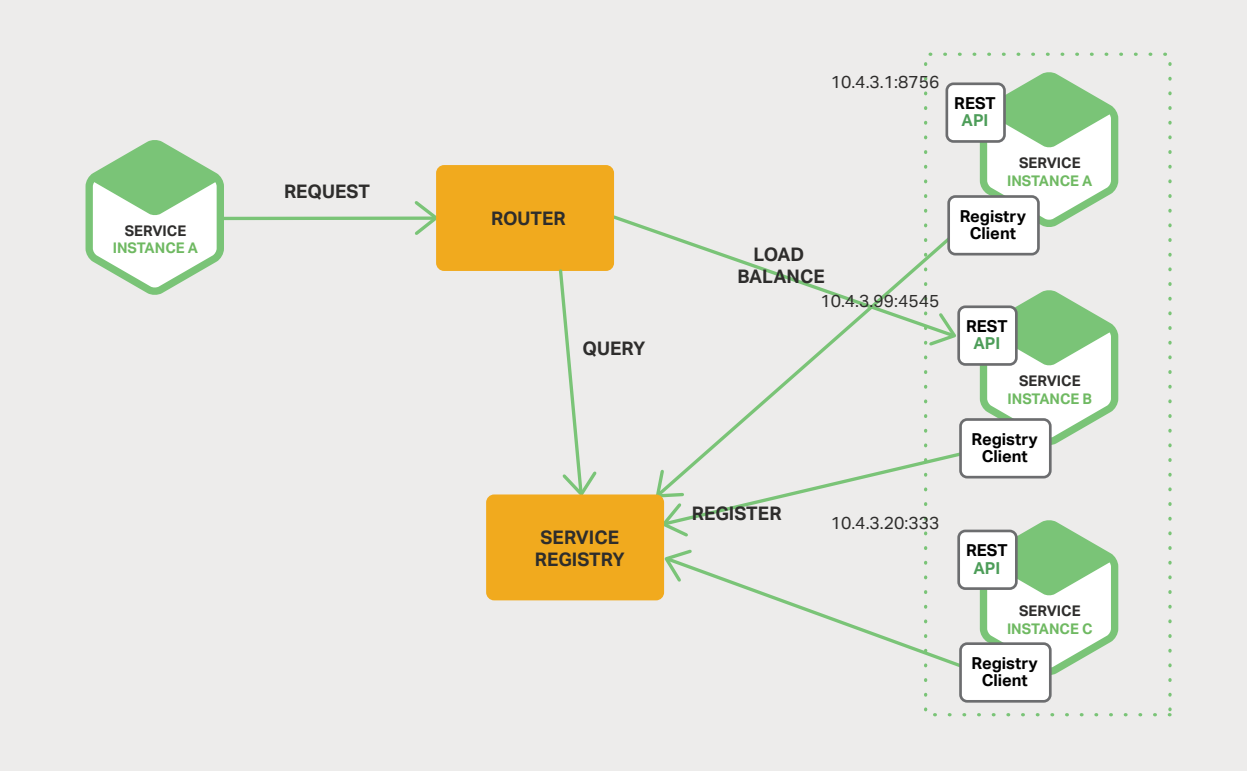


Figure 4-3. Service discovery can also be handled among servers.

The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. As with client-side discovery, service instances are registered and deregistered with the service registry.

The AWS Elastic Load Balancer (ELB) is an example of a server-side discovery router. ELB is commonly used to load balance external traffic from the Internet. However, you can also use ELB to load balance traffic that is internal to a virtual private cloud (VPC).

A client makes requests (HTTP or TCP) via the ELB using its DNS name. The ELB load balances the traffic among a set of registered Elastic Compute Cloud (EC2) instances or EC2 Container Service (ECS) containers. There isn't a separately visible service registry. Instead, EC2 instances and ECS containers are registered with the ELB itself.

HTTP servers and load balancers such as NGINX Plus and NGINX can also be used as a server-side discovery load balancer. For example, this blog post describes using Consul Template to dynamically reconfigure NGINX reverse proxying. Consul Template is a tool that periodically regenerates arbitrary configuration files from configuration data stored in the Consul service registry. It runs an arbitrary shell command whenever the files change. In the example described in the blog post, Consul Template generates an **nginx.conf** file, which configures the reverse proxying, and then runs a command that tells NGINX to reload the configuration. A more sophisticated implementation could dynamically reconfigure NGINX Plus using either its HTTP API or DNS.

Some deployment environments such as Kubernetes and Marathon run a proxy on each host in the cluster. The proxy plays the role of a server-side discovery load balancer. In order to make a request to a service, a client routes the request via the proxy using the host's IP address and the service's assigned port. The proxy then transparently forwards the request to an available service instance running somewhere in the cluster.

The server-side discovery pattern has several benefits and drawbacks. One great benefit of this pattern is that details of discovery are abstracted away from the client. Clients simply make requests to the load balancer. This eliminates the need to implement discovery logic for each programming language and framework used by your service clients. Also, as mentioned above, some deployment environments provide this functionality for free. This pattern also has some drawbacks, however. Unless the load balancer is provided by the deployment environment, it is yet another highly available system component that you need to set up and manage.

The Service Registry

The service registry is a key part of service discovery. It is a database containing the network locations of service instances. A service registry needs to be highly available and up to date. Clients can cache network locations obtained from the service registry. However, that information eventually becomes out of date and clients become unable to discover service instances. Consequently, a service registry consists of a cluster of servers that use a replication protocol to maintain consistency.

As mentioned earlier, Netflix Eureka is good example of a service registry. It provides a REST API for registering and querying service instances. A service instance registers its network location using a `POST` request. Every 30 seconds it must refresh its registration using a `PUT` request. A registration is removed by either using an HTTP `DELETE` request or by the instance registration timing out. As you might expect, a client can retrieve the registered service instances by using an HTTP `GET` request.

Netflix achieves high availability by running one or more Eureka servers in each Amazon EC2 availability zone. Each Eureka server runs on an EC2 instance that has an Elastic IP address. DNS `TEXT` records are used to store the Eureka cluster configuration, which is a map from availability zones to a list of the network locations of Eureka servers. When a Eureka server starts up, it queries DNS to retrieve the Eureka cluster configuration, locates its peers, and assigns itself an unused Elastic IP address.

Eureka clients – services and service clients – query DNS to discover the network locations of Eureka servers. Clients prefer to use a Eureka server in the same availability zone. However, if none is available, the client uses a Eureka server in another availability zone.

Other examples of service registries include:

- etcd – A highly available, distributed, consistent, key-value store that is used for shared configuration and service discovery. Two notable projects that use etcd are Kubernetes and Cloud Foundry.
- Consul – A tool for discovering and configuring services. It provides an API that allows clients to register and discover services. Consul can perform health checks to determine service availability.
- Apache ZooKeeper – A widely used, high-performance coordination service for distributed applications. Apache ZooKeeper was originally a subproject of Hadoop, but is now a separate, top-level project.

Also, as noted previously, some systems such as Kubernetes, Marathon, and AWS do not have an explicit service registry. Instead, the service registry is just a built-in part of the infrastructure.

Now that we have looked at the concept of a service registry, let's look at how service instances are registered with the service registry.

Service Registration Options

As previously mentioned, service instances must be registered with and unregistered from the service registry. There are a couple of different ways to handle the registration and unregistration. One option is for service instances to register themselves, the self-registration pattern. The other option is for some other system component to manage the registration of service instances, the third-party registration pattern. Let's first look at the self-registration pattern.

The Self-Registration Pattern

When using the self-registration pattern, a service instance is responsible for registering and unregistering itself with the service registry. Also, if required, a service instance sends heartbeat requests to prevent its registration from expiring.

Figure 4-4 shows the structure of this pattern.

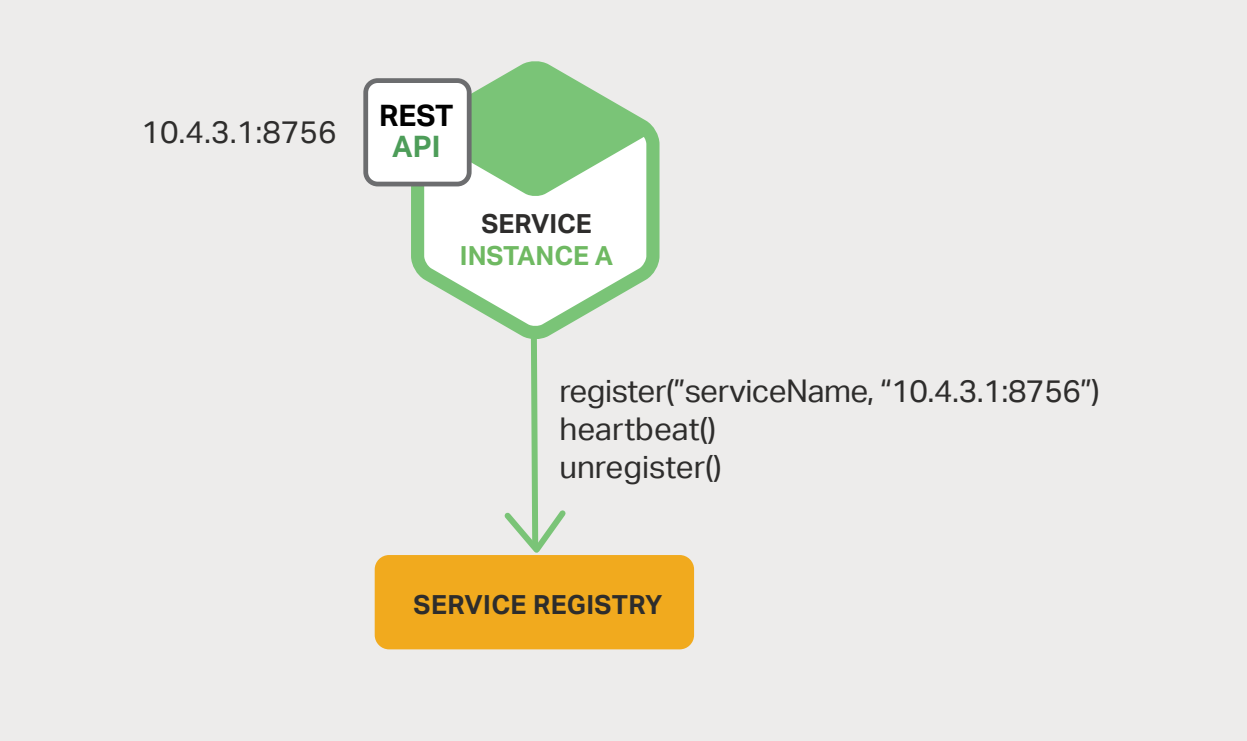


Figure 4-4. Services can handle their own registration.

A good example of this approach is the Netflix OSS Eureka client. The Eureka client handles all aspects of service instance registration and unregistration. The Spring Cloud project, which implements various patterns including service discovery, makes it easy to automatically register a service instance with Eureka. You simply annotate your Java Configuration class with an `@EnableEurekaClient` annotation.

The self-registration pattern has various benefits and drawbacks. One benefit is that it is relatively simple and doesn't require any other system components. However, a major drawback is that it couples the service instances to the service registry. You must implement the registration code in each programming language and framework used by your services.

The alternative approach, which decouples services from the service registry, is the third-party registration pattern.

The Third-Party Registration Pattern

When using the third-party registration pattern, service instances aren't responsible for registering themselves with the service registry. Instead, another system component known as the *service registrar* handles the registration. The service registrar tracks changes to the set of running instances by either polling the deployment environment or subscribing to events. When it notices a newly available service instance, it registers the instance with the service registry. The service registrar also unregisters terminated service instances.

Figure 4-5 shows the structure of this pattern:

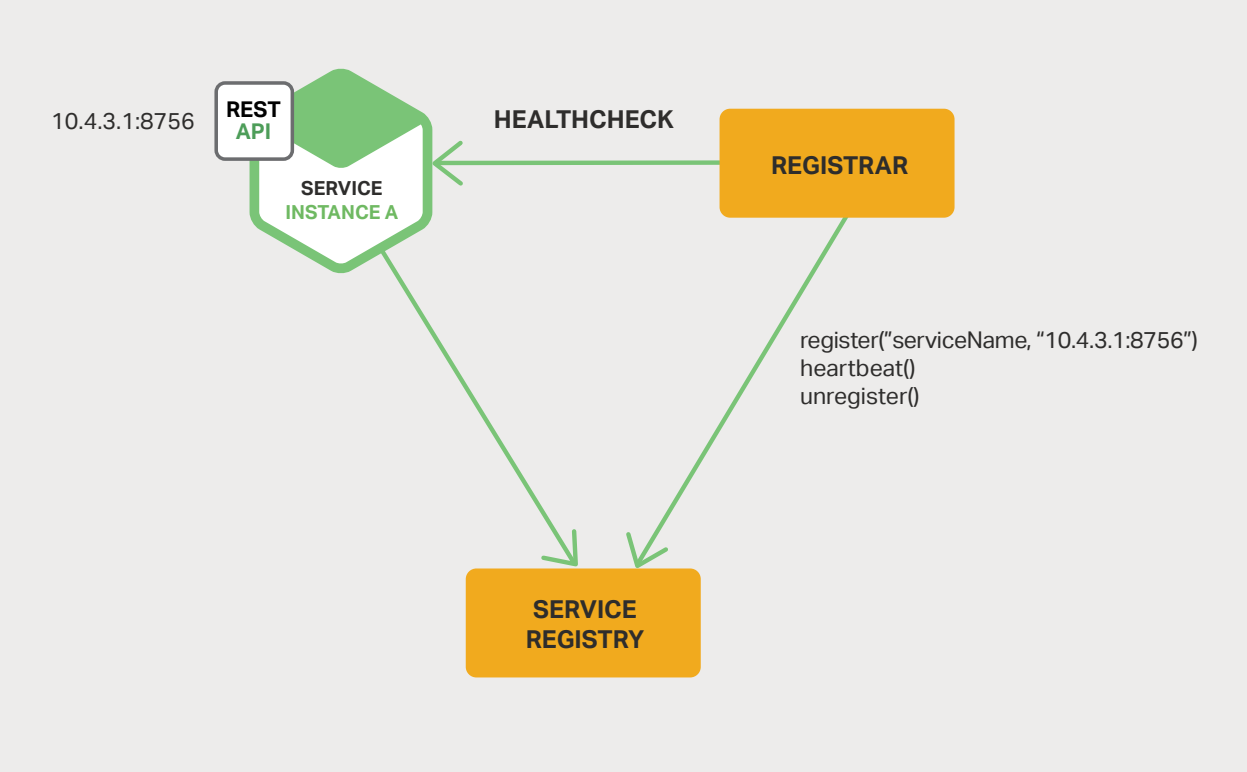


Figure 4-5. A separate registrar service can be responsible for registering others.

One example of a service registrar is the open source Registrator project. It automatically registers and unregisters service instances that are deployed as Docker containers. Registrator supports several service registries, including etcd and Consul.

Another example of a service registrar is NetflixOSS Prana. Primarily intended for services written in non-JVM languages, it is a sidecar application that runs side by side with a service instance. Prana registers and unregisters the service instance with Netflix Eureka.

The service registrar is a built-in component in some deployment environments. The EC2 instances created by an Autoscaling Group can be automatically registered with an ELB. Kubernetes services are automatically registered and made available for discovery.

The third-party registration pattern has various benefits and drawbacks. A major benefit is that services are decoupled from the service registry. You don't need to implement service-registration logic for each programming language and framework used by your developers. Instead, service instance registration is handled in a centralized manner within a dedicated service.

One drawback of this pattern is that unless it's built into the deployment environment, it is yet another highly available system component that you need to set up and manage.

Summary

In a microservices application, the set of running service instances changes dynamically. Instances have dynamically assigned network locations. Consequently, in order for a client to make a request to a service it must use a service-discovery mechanism.

A key part of service discovery is the service registry. The service registry is a database of available service instances. The service registry provides a management API and a query API. Service instances are registered with and unregistered from the service registry using the management API. The query API is used by system components to discover available service instances.

There are two main service-discovery patterns: client-side discovery and service-side discovery. In systems that use client-side service discovery, clients query the service registry, select an available instance, and make a request. In systems that use server-side discovery, clients make requests via a router, which queries the service registry and forwards the request to an available instance.

There are two main ways that service instances are registered with and unregistered from the service registry. One option is for service instances to register themselves with the service registry, the self-registration pattern. The other option is for some other system component to handle the registration and unregistration on behalf of the service, the third-party registration pattern.

In some deployment environments you need to set up your own service-discovery infrastructure using a service registry such as Netflix Eureka, etcd, or Apache ZooKeeper. In other deployment environments, service discovery is built in. For example, Kubernetes and Marathon handle service instance registration and unregistration. They also run a proxy on each cluster host that plays the role of server-side discovery router.

An HTTP reverse proxy and load balancer such as NGINX can also be used as a server-side discovery load balancer. The service registry can push the routing information to NGINX and invoke a graceful configuration update; for example, you can use Consul Template. NGINX Plus supports additional dynamic reconfiguration mechanisms – it can pull information about service instances from the registry using DNS, and it provides an API for remote reconfiguration.

Microservices in Action: NGINX Flexibility

by Floyd Smith

In a microservices environment, your backend infrastructure is likely to be constantly changing as services are created, deployed, and scaled up and down as a result of autoscaling, failures, and upgrades. As described in this chapter, a service discovery mechanism is required in environments where service locations are dynamically reassigned.

Part of the benefit of using NGINX for microservices is that you can easily configure it to automatically react to changes in backend infrastructure. NGINX configuration is not only easy and flexible, it's also compatible with the use of templates, as used in Amazon Web Services, making it easier to manage changes for a specific service and to manage changing sets of services subject to load balancing.

NGINX Plus features an on-the-fly reconfiguration API, eliminating the need to restart NGINX Plus or manually reload its configuration to get it to recognize changes to the set of services being load balanced. In NGINX Plus Release 8 and later, the changes you make with the API can be configured to persist across restarts and configuration reloads. (Reloads do not require a restart and do not drop connections.) And NGINX Plus Release 9 and later have support for service discovery using DNS `SRV` records, enabling tighter integration with existing server discovery platforms, such as Consul and etcd.

We here at NGINX have created a model for managing service discovery:

1. Run separate Docker containers for each of several apps, including a service discovery app such as etcd, a service registration tool, one or more backend servers, and NGINX Plus itself to load balance the other containers.
2. The registration tool monitors Docker for new containers and registers new services with the service discovery tool, also removing containers that disappear.
3. Containers and the services they run are automatically added to or removed from the group of load-balanced upstream servers.

Demo apps for this process are available for several service-discovery apps: Consul APIs, DNS `SRV` records from Consul, etcd, and ZooKeeper.

5 Event-Driven Data Management for Microservices

This is the fifth chapter of this ebook about building applications with microservices. The first chapter introduces the Microservices Architecture pattern and discusses the benefits and drawbacks of using microservices. The second and third describe different aspects of communication within a microservices architecture. The fourth chapter explores the closely related problem of service discovery. In this chapter, we change gears and look at the distributed data management problems that arise in a microservices architecture.

Microservices and the Problem of Distributed Data Management

A monolithic application typically has a single relational database. A key benefit of using a relational database is that your application can use ACID transactions, which provide some important guarantees:

- Atomicity – Changes are made atomically
- Consistency – The state of the database is always consistent
- Isolation – Even though transactions are executed concurrently, it appears they are executed serially
- Durable – Once a transaction has committed, it is not undone

As a result, your application can simply begin a transaction, change (insert, update, and delete) multiple rows, and commit the transaction.

Another great benefit of using a relational database is that it provides SQL, which is a rich, declarative, and standardized query language. You can easily write a query that combines data from multiple tables. The RDBMS query planner then determines the optimal way to execute the query. You don't have to worry about low-level details such as how to access the database. And, because all of your application's data is in one database, it is easy to query.

Unfortunately, data access becomes much more complex when we move to a microservices architecture. That is because the data owned by each microservice is private to that microservice and can only be accessed via its API. Encapsulating the data ensures that the microservices are loosely coupled and can evolve independently of one another. If multiple services access the same data, schema updates require time-consuming, coordinated updates to all of the services.

To make matters worse, different microservices often use different kinds of databases. Modern applications store and process diverse kinds of data, and a relational database is not always the best choice. For some use cases, a particular NoSQL database might have a more convenient data model and offer much better performance and scalability. For example, it makes sense for a service that stores and queries text to use a text search engine such as Elasticsearch. Similarly, a service that stores social graph data should probably use a graph database, such as Neo4j. Consequently, microservices-based applications often use a mixture of SQL and NoSQL databases, the so-called polyglot persistence approach.

A partitioned, polyglot-persistent architecture for data storage has many benefits, including loosely coupled services and better performance and scalability. However, it does introduce some distributed data management challenges.

The first challenge is how to implement business transactions that maintain consistency across multiple services. To see why this is a problem, let's take a look at an example of an online B2B store. The Customer Service maintains information about customers, including their credit lines. The Order Service manages orders and must verify that a new order doesn't violate the customer's credit limit. In the monolithic version of this application, the Order Service can simply use an ACID transaction to check the available credit and create the order.

In contrast, in a microservices architecture the ORDER and CUSTOMER tables are private to their respective services, as shown in Figure 5-1:

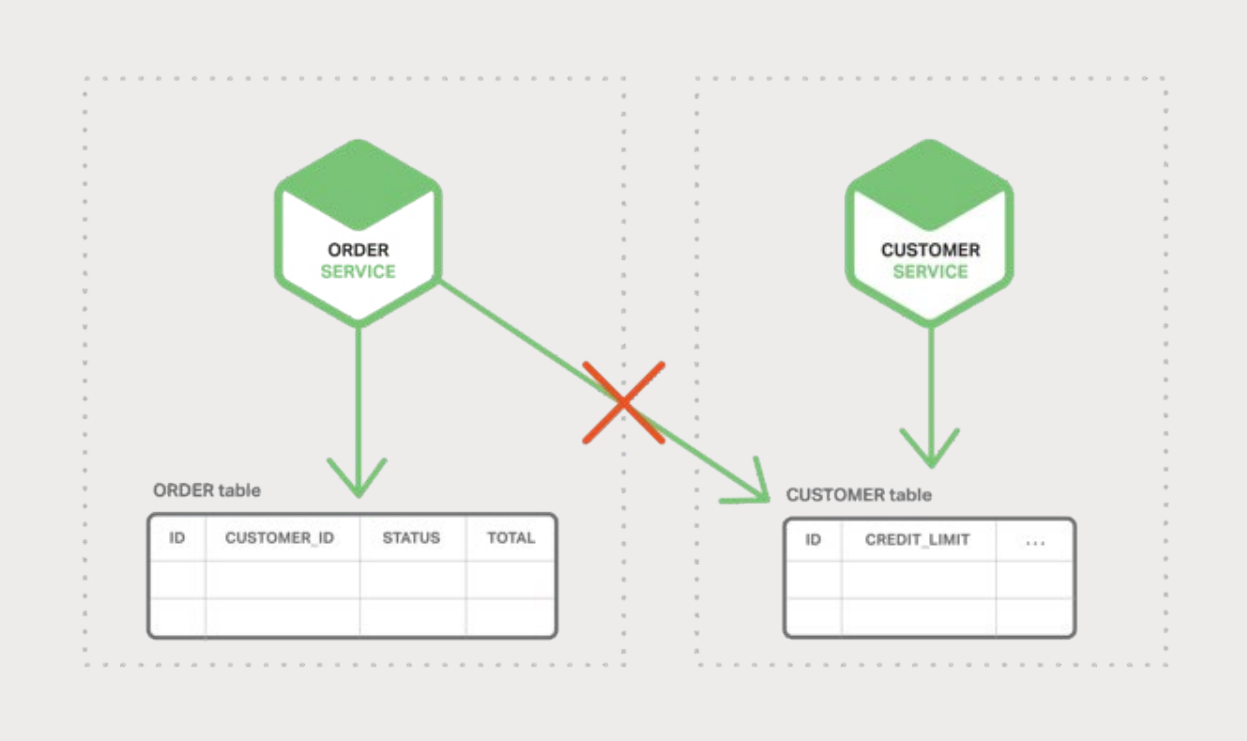


Figure 5-1. Microservices each have their own data.

The Order Service cannot access the CUSTOMER table directly. It can only use the API provided by the Customer Service. The Order Service could potentially use distributed transactions, also known as two-phase commit (2PC). However, 2PC is usually not a viable option in modern applications. The CAP theorem requires you to choose between availability and ACID-style consistency, and availability is usually the better choice. Moreover, many modern technologies, such as most NoSQL databases, do not support 2PC. Maintaining data consistency across services and databases is essential, so we need another solution.

The second challenge is how to implement queries that retrieve data from multiple services. For example, let's imagine that the application needs to display a customer and his recent orders. If the Order Service provides an API for retrieving a customer's orders then you can retrieve this data using an application-side join. The application retrieves the customer from the Customer Service and the customer's orders from the Order Service. Suppose, however, that the Order Service only supports the lookup of orders by their primary key (perhaps it uses a NoSQL database that only supports primary key-based retrievals). In this situation, there is no obvious way to retrieve the needed data.

Event-Driven Architecture

For many applications, the solution is to use an event-driven architecture. In this architecture, a microservice publishes an event when something notable happens, such as when it updates a business entity. Other microservices subscribe to those events. When a microservice receives an event it can update its own business entities, which might lead to more events being published.

You can use events to implement business transactions that span multiple services. A transaction consists of a series of steps. Each step consists of a microservice updating a business entity and publishing an event that triggers the next step. The following sequence of diagrams shows how you can use an event-driven approach to checking for available credit when creating an order.

The microservices exchange events via a Message Broker:

- The Order Service creates an Order with status NEW and publishes an Order Created event.

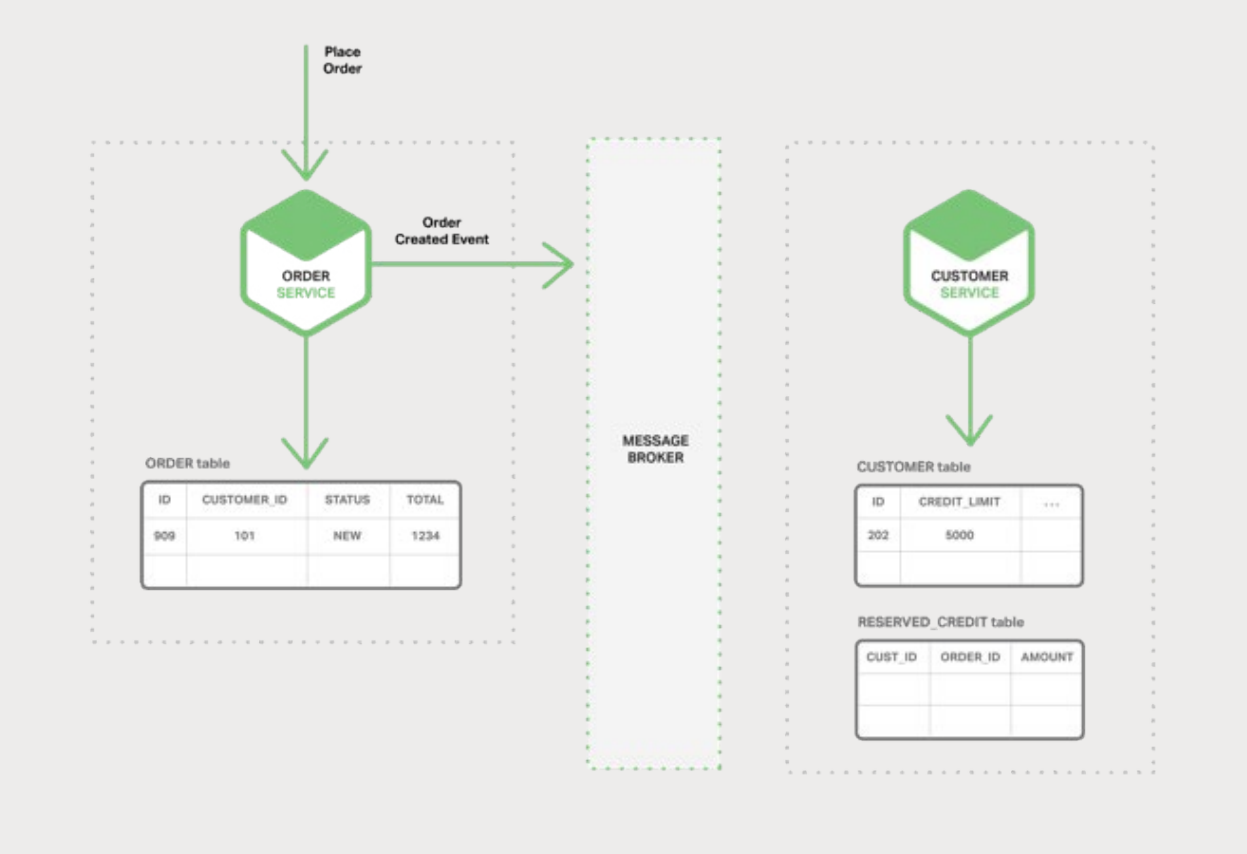


Figure 5-2. The Order Service publishes an event.

- The Customer Service consumes the Order Created event, reserves credit for the order, and publishes a Credit Reserved event.

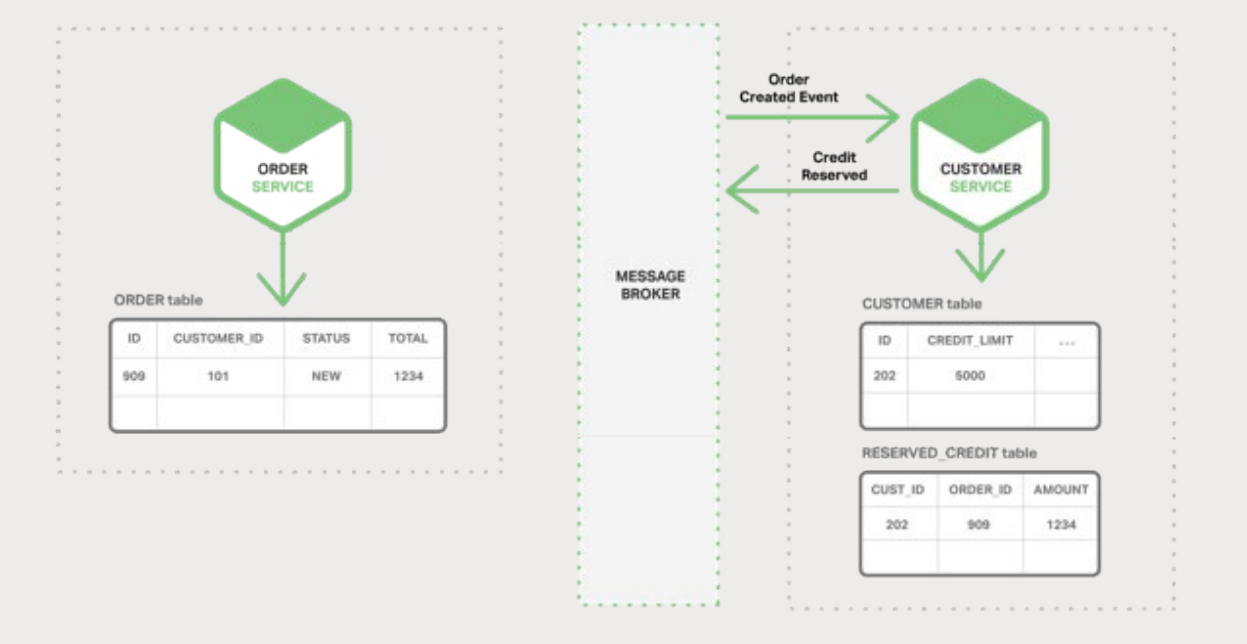


Figure 5-3. The Customer Service responds.

- The Order Service consumes the Credit Reserved event and changes the status of the order to OPEN.

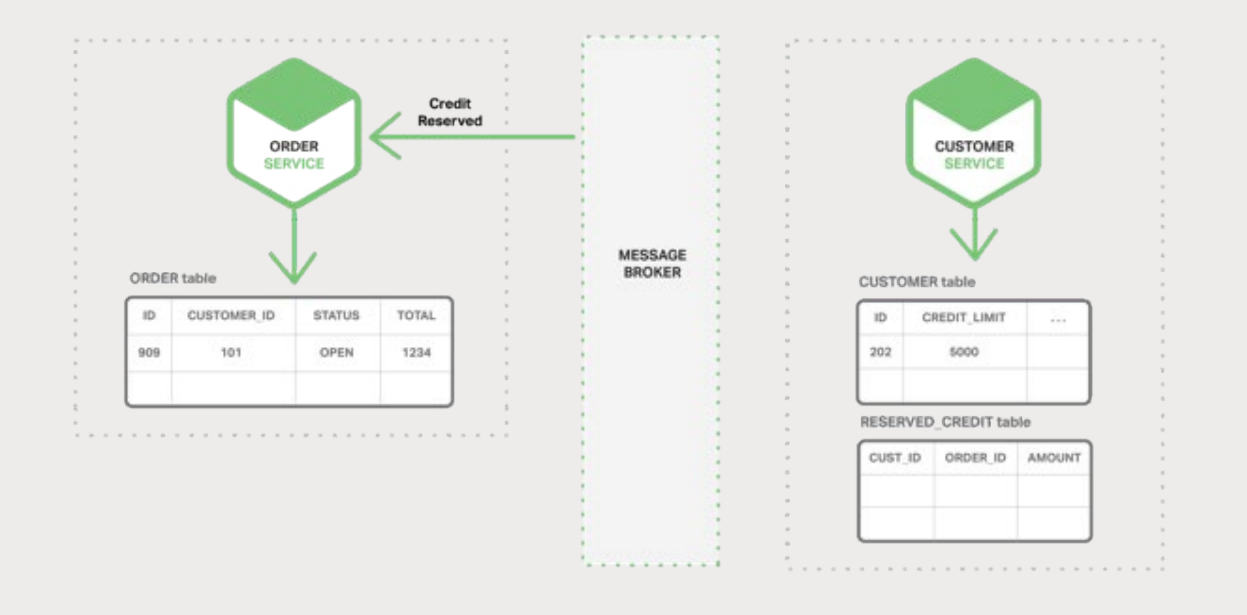


Figure 5-4. The Order Service acts on the response.

A more complex scenario could involve additional steps, such as reserving inventory at the same time as the customer's credit is checked.

Provided that (a) each service atomically updates the database and publishes an event – more on that later – and (b) the Message Broker guarantees that events are delivered at least once, then you can implement business transactions that span multiple services. It is important to note that these are not ACID transactions. They offer much weaker guarantees such as eventual consistency. This transaction model has been referred to as the BASE model.

You can also use events to maintain materialized views that pre-join data owned by multiple microservices. The service that maintains the view subscribes to the relevant events and updates the view. Figure 5-5 depicts a Customer Order View Updater Service that updates the Customer Order View based on events published by the Customer Service and Order Service.

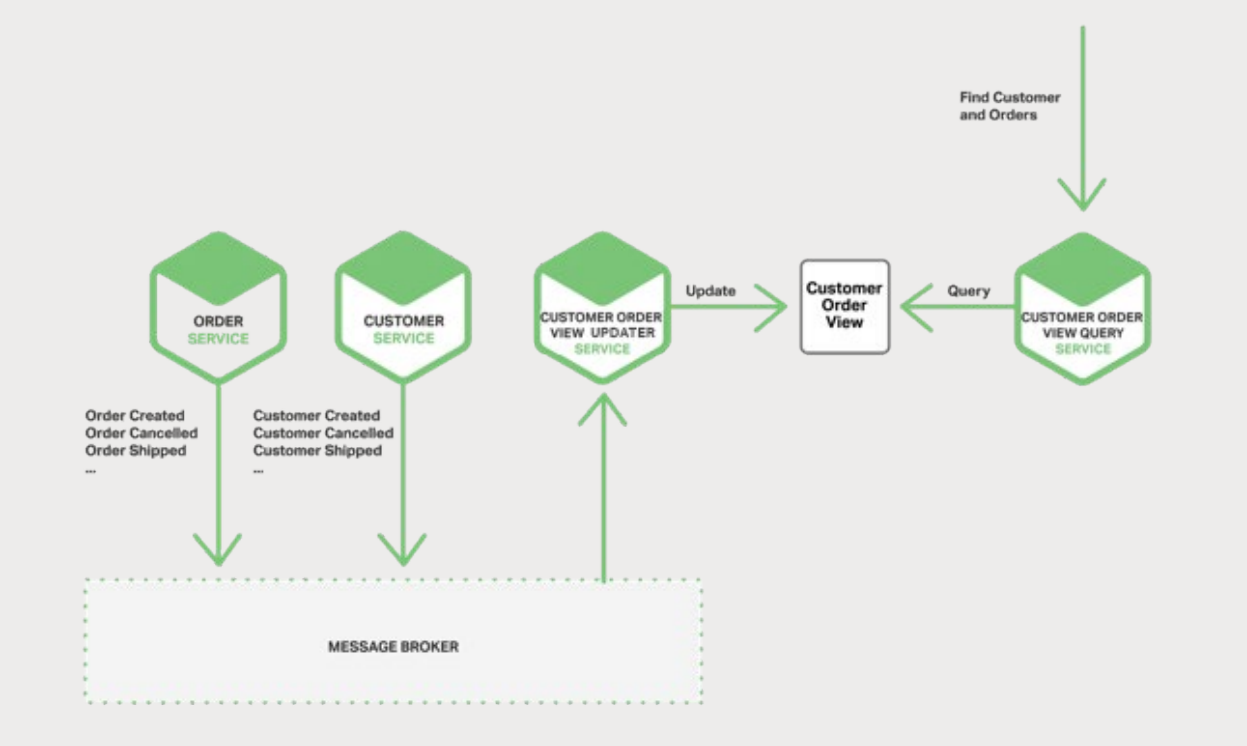


Figure 5-5. The Customer Order View is accessed by two services.

When the Customer Order View Updater Service receives a Customer or Order event, it updates the Customer Order View datastore. You could implement the Customer Order View using a document database such as MongoDB and store one document for each Customer. The Customer Order View Query Service handles requests for a customer and recent orders by querying the Customer Order View datastore.

An event-driven architecture has several benefits and drawbacks. It enables the implementation of transactions that span multiple services and provide eventual consistency. Another benefit is that it also enables an application to maintain materialized views.

One drawback is that the programming model is more complex than when using ACID transactions. Often you must implement compensating transactions to recover from application-level failures; for example, you must cancel an order if the credit check fails. Also, applications must deal with inconsistent data. That is because changes made by in-flight transactions are visible. The application can also see inconsistencies if it reads from a materialized view that is not yet updated. Another drawback is that subscribers must detect and ignore duplicate events.

Achieving Atomicity

In an event-driven architecture there is also the problem of atomically updating the database and publishing an event. For example, the Order Service must insert a row into the ORDER table and publish an Order Created event. It is essential that these two operations are done atomically. If the service crashes after updating the database but before publishing the event, the system becomes inconsistent. The standard way to ensure atomicity is to use a distributed transaction involving the database and the Message Broker. However, for the reasons described above, such as the CAP theorem, this is exactly what we do not want to do.

Publishing Events Using Local Transactions

One way to achieve atomicity is for the application to publish events using a multi-step process involving only local transactions. The trick is to have an EVENT table, which functions as a message queue, in the database that stores the state of the business entities. The application begins a (local) database transaction, updates the state of the business entities, inserts an event into the EVENT table, and commits the transaction. A separate application thread or process queries the EVENT table, publishes the events to the Message Broker, and then uses a local transaction to mark the events as published. Figure 5-6 shows the design.

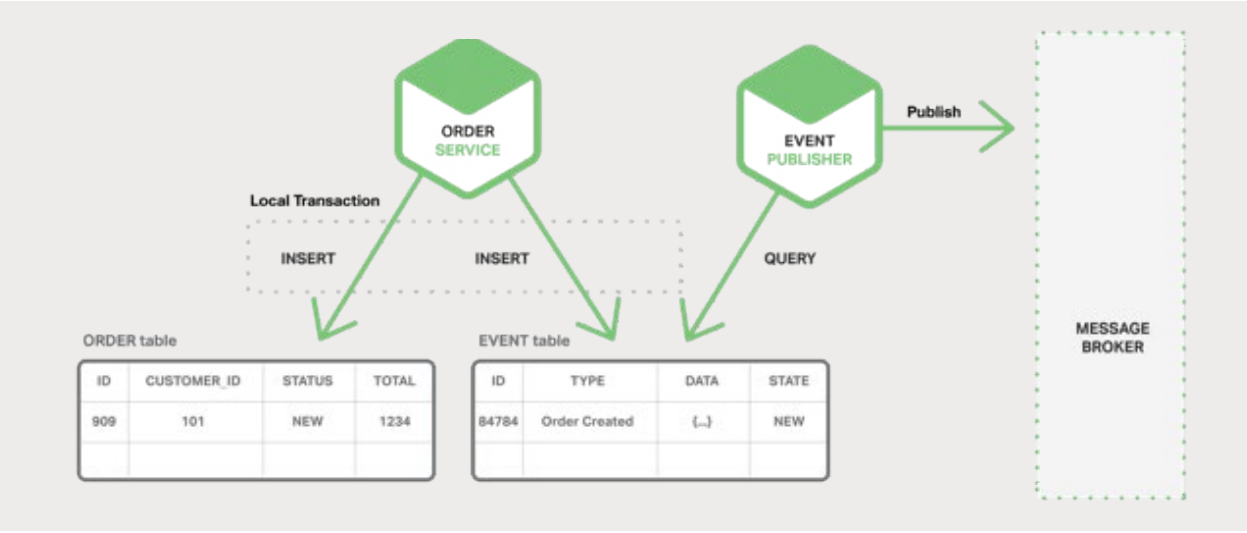


Figure 5-6. Achieving atomicity with local transactions.

The Order Service inserts a row into the ORDER table and inserts an Order Created event into the EVENT table. The Event Publisher thread or process queries the EVENT table for unpublished events, publishes the events, and then updates the EVENT table to mark the events as published.

This approach has several benefits and drawbacks. One benefit is that it guarantees an event is published for each update without relying on 2PC. Also, the application publishes business-level events, which eliminates the need to infer them. One drawback of this approach is that it is potentially error-prone since the developer must remember to publish events. A limitation of this approach is that it is challenging to implement when using some NoSQL databases because of their limited transaction and query capabilities.

This approach eliminates the need for 2PC by having the application use local transactions to update state and publish events. Let's now look at an approach that achieves atomicity by having the application simply update state.

Mining a Database Transaction Log

Another way to achieve atomicity without 2PC is for the events to be published by a thread or process that mines the database's transaction or commit log. The application updates the database, so changes are recorded in the database's transaction log. The Transaction Log Miner thread or process reads the transaction log and publishes events to the Message Broker. Figure 5-7 shows the design.

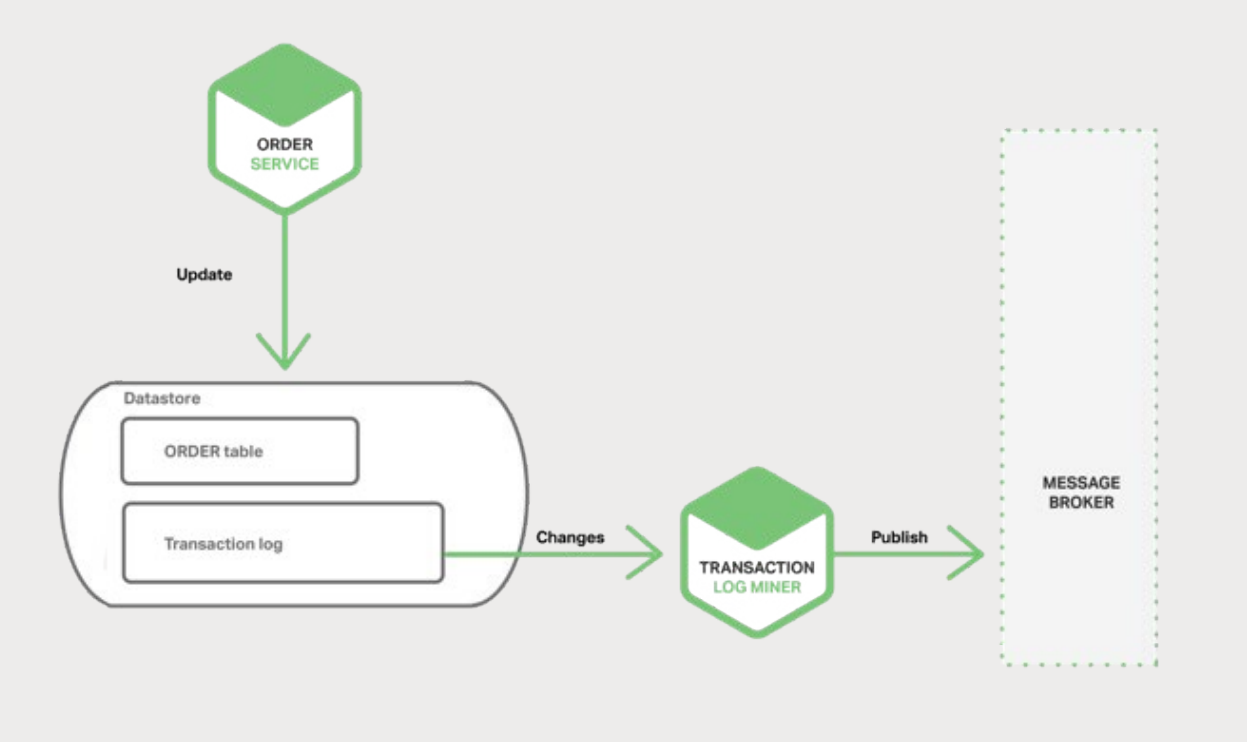


Figure 5-7. A Message Broker can arbitrate data transactions.

An example of this approach is the open source LinkedIn Databus project. Databus mines the Oracle transaction log and publishes events corresponding to the changes. LinkedIn uses Databus to keep various derived data stores consistent with the system of record.

Another example is the streams mechanism in AWS DynamoDB, which is a managed NoSQL database. A DynamoDB stream contains the time-ordered sequence of changes (create, update, and delete operations) made to the items in a DynamoDB table in the last 24 hours. An application can read those changes from the stream and, for example, publish them as events.

Transaction log mining has various benefits and drawbacks. One benefit is that it guarantees that an event is published for each update without using 2PC. Transaction log mining can also simplify the application by separating event publishing from the application's business logic. A major drawback is that the format of the transaction log is proprietary to each database and can even change between database versions. Also, it can be difficult to reverse engineer the high-level business events from the low-level updates recorded in the transaction log.

Transaction log mining eliminates the need for 2PC by having the application do one thing: update the database. Let's now look at a different approach that eliminates the updates and relies solely on events.

Using Event Sourcing

Event sourcing achieves atomicity without 2PC by using a radically different, event-centric approach to persisting business entities. Rather than store the current state of an entity, the application stores a sequence of state-changing events. The application reconstructs an entity's current state by replaying the events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic.

To see how event sourcing works, consider the Order entity as an example. In a traditional approach, each order maps to a row in an ORDER table and to rows in, for example, an ORDER_LINE_ITEM table.

But when using event sourcing, the Order Service stores an Order in the form of its state-changing events: Created, Approved, Shipped, Cancelled. Each event contains sufficient data to reconstruct the Order's state.

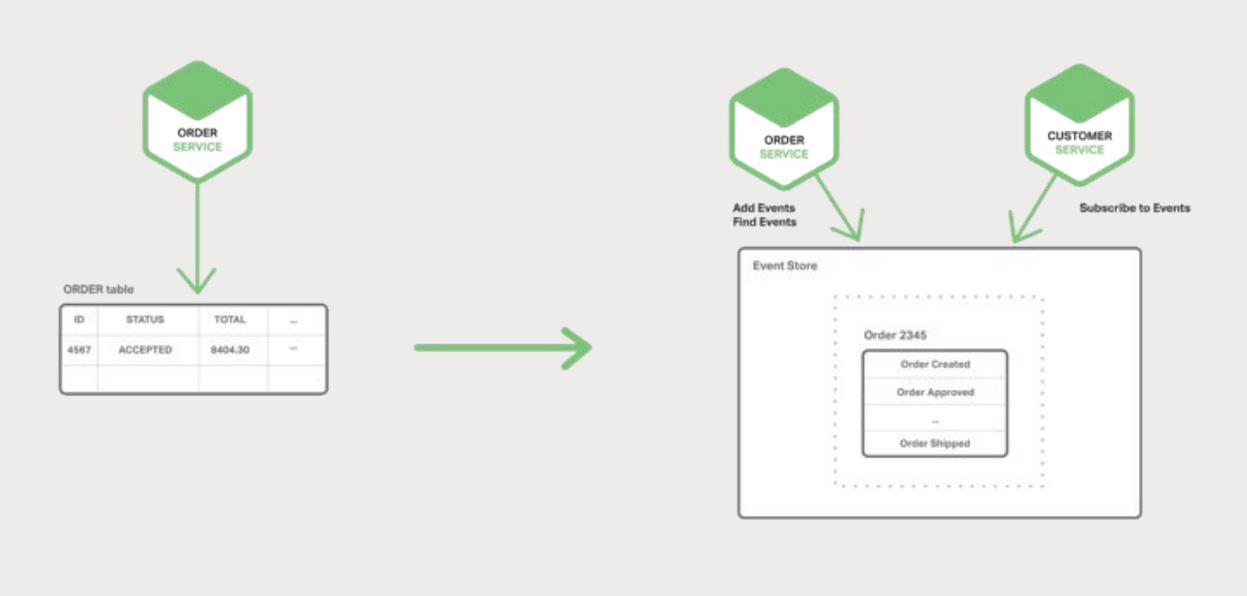


Figure 5-8. Events can have complete recovery data.

Events persist in an Event Store, which is a database of events. The store has an API for adding and retrieving an entity's events. The Event Store also behaves like the Message Broker in the architectures we described previously. It provides an API that enables services to subscribe to events. The Event Store delivers all events to all interested subscribers. The Event Store is the backbone of an event-driven microservices architecture.

Event sourcing has several benefits. It solves one of the key problems in implementing an event-driven architecture and makes it possible to reliably publish events whenever state changes. As a result, it solves data consistency issues in a microservices architecture. Also, because it persists events rather than domain objects, it mostly avoids the object-relational impedance mismatch problem. Event sourcing also provides a 100% reliable audit log of the changes made to a business entity and makes it possible to implement temporal queries that determine the state of an entity at any point in time. Another major benefit of event sourcing is that your business logic consists of loosely coupled business entities that exchange events. This makes it a lot easier to migrate from a monolithic application to a microservices architecture.

Event sourcing also has some drawbacks. It is a different and unfamiliar style of programming and so there is a learning curve. The event store only directly supports the lookup of business entities by primary key. You must use command query responsibility separation (CQRS) to implement queries. As a result, applications must handle eventually consistent data.

Summary

In a microservices architecture, each microservice has its own private datastore. Different microservices might use different SQL and NoSQL databases. While this database architecture has significant benefits, it creates some distributed data management challenges. The first challenge is how to implement business transactions that maintain consistency across multiple services. The second challenge is how to implement queries that retrieve data from multiple services.

For many applications, the solution is to use an event-driven architecture. One challenge with implementing an event-driven architecture is how to atomically update state and how to publish events. There are a few ways to accomplish this, including using the database as a message queue, transaction log mining, and event sourcing.

Microservices in Action: NGINX and Storage Optimization

by Floyd Smith

A microservices-based approach to storage involves a greater number and variety of data stores, more complexity in how you access and update data, and greater challenges for both Dev and Ops in maintaining data consistency. NGINX provides crucial support for this kind of data management, in three main areas:

1. **Caching and microcaching of data** – Caching static files and microcaching application-generated content with NGINX reduces the load on your application, increasing performance and reducing the potential for problems.
2. **Flexibility and scalability per data store** – Once you implement NGINX as a reverse proxy server, your apps gain great flexibility in creating, sizing, running, and resizing data storage servers to meet changing requirements – vital when every service has its own data store.
3. **Monitoring and management of services, including data services** – With the number of data servers multiplying, supporting complex operations is critical, as are monitoring and management tools. NGINX Plus has built-in tools and interfaces to application performance management partners such as Data Dog, Dynatrace, and New Relic.

Examples of microservice-specific data management are included in the three Models of the NGINX Microservices Reference Architecture, giving you a starting point for your own design decisions and implementation.

6 Choosing a Microservices Deployment Strategy

This is the sixth chapter in this ebook about building applications with microservices. Chapter 1 introduces the Microservices Architecture pattern and discusses the benefits and drawbacks of using microservices. The following chapters discuss different aspects of the microservices architecture: using an API Gateway, inter-process communication, service discovery, and event-driven data management. In this chapter, we look at strategies for deploying microservices.

Motivations

Deploying a monolithic application means running one or more identical copies of a single, usually large, application. You typically provision N servers (physical or virtual) and run M instances of the application on each server. The deployment of a monolithic application is not always entirely straightforward, but it is much simpler than deploying a microservices application.

A microservices application consists of tens or even hundreds of services. Services are written in a variety of languages and frameworks. Each one is a mini-application with its own specific deployment, resource, scaling, and monitoring requirements. For example, you need to run a certain number of instances of each service based on the demand for that service. Also, each service instance must be provided with the appropriate CPU, memory, and I/O resources. What is even more challenging is that despite this complexity, deploying services must be fast, reliable and cost-effective.

There are a few different microservice deployment patterns. Let's look first at the Multiple Service Instances per Host pattern.

Multiple Service Instances Per Host Pattern

One way to deploy your microservices is to use the Multiple Service Instances per Host pattern. When using this pattern, you provision one or more physical or virtual hosts and run multiple service instances on each one. In many ways, this is the traditional approach to application deployment. Each service instance runs at a well-known port on one or more hosts. The host machines are commonly treated like pets.

Figure 6-1 shows the structure of this pattern:

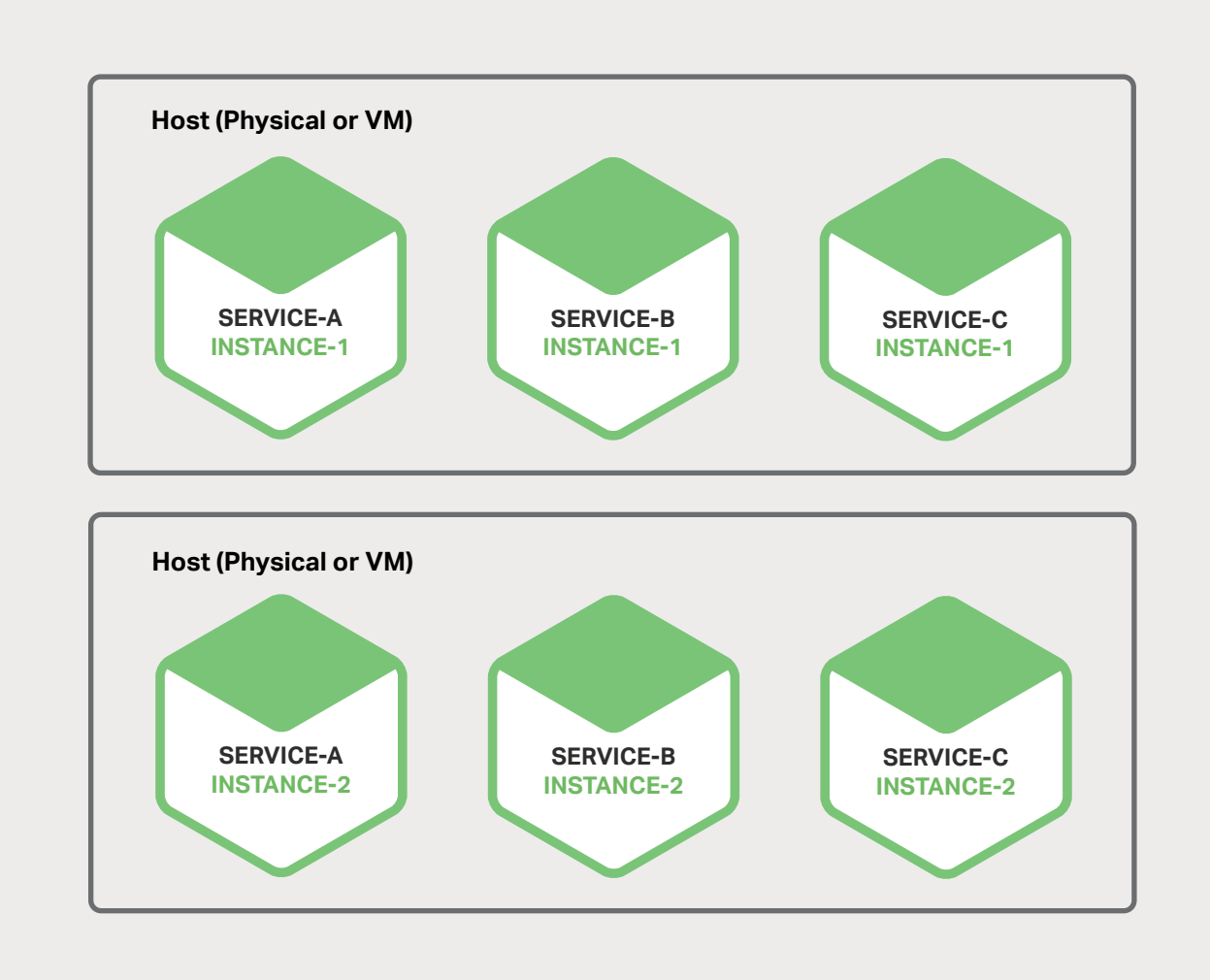


Figure 6-1. Hosts can each support multiple service instances.

There are a couple of variants of this pattern. One variant is for each service instance to be a process or a process group. For example, you might deploy a Java service instance as a web application on an Apache Tomcat server. A Node.js service instance might consist of a parent process and one or more child processes.

The other variant of this pattern is to run multiple service instances in the same process or process group. For example, you could deploy multiple Java web applications on the same Apache Tomcat server or run multiple OSGI bundles in the same OSGI container.

The Multiple Service Instances per Host pattern has both benefits and drawbacks. One major benefit is its resource usage is relatively efficient. Multiple service instances share the server and its operating system. It's even more efficient if a process or group runs multiple service instances, for example, multiple web applications sharing the same Apache Tomcat server and JVM.

Another benefit of this pattern is that deploying a service instance is relatively fast. You simply copy the service to a host and start it. If the service is written in Java, you copy a JAR or WAR file. For other languages, such as Node.js or Ruby, you copy the source code. In either case, the number of bytes copied over the network is relatively small.

Also, because of the lack of overhead, starting a service is usually very fast. If the service is its own process, you simply start it. Otherwise, if the service is one of several instances running in the same container process or process group, you either dynamically deploy it into the container or restart the container.

Despite its appeal, the Multiple Service Instances per Host pattern has some significant drawbacks. One major drawback is that there is little or no isolation of the service instances, unless each service instance is a separate process. While you can accurately monitor each service instance's resource utilization, you cannot limit the resources each instance uses. It's possible for a misbehaving service instance to consume all of the memory or CPU of the host.

There is no isolation at all if multiple service instances run in the same process. All instances might, for example, share the same JVM heap. A misbehaving service instance could easily break the other services running in the same process. Moreover, you have no way to monitor the resources used by each service instance.

Another significant problem with this approach is that the operations team that deploys a service has to know the specific details of how to do it. Services can be written in a variety of languages and frameworks, so there are lots of details that the development team must share with operations. This complexity increases the risk of errors during deployment.

As you can see, despite its familiarity, the Multiple Service Instances per Host pattern has some significant drawbacks. Let's now look at other ways of deploying microservices that avoid these problems.

Service Instance per Host Pattern

Another way to deploy your microservices is the Service Instance per Host pattern. When you use this pattern, you run each service instance in isolation on its own host. There are two different different specializations of this pattern: Service Instance per Virtual Machine and Service Instance per Container.

Service Instance per Virtual Machine Pattern

When you use Service Instance per Virtual Machine pattern, you package each service as a virtual machine (VM) image such as an Amazon EC2 AMI. Each service instance is a VM (for example, an EC2 instance) that is launched using that VM image.

Figure 6-2 shows the structure of this pattern:

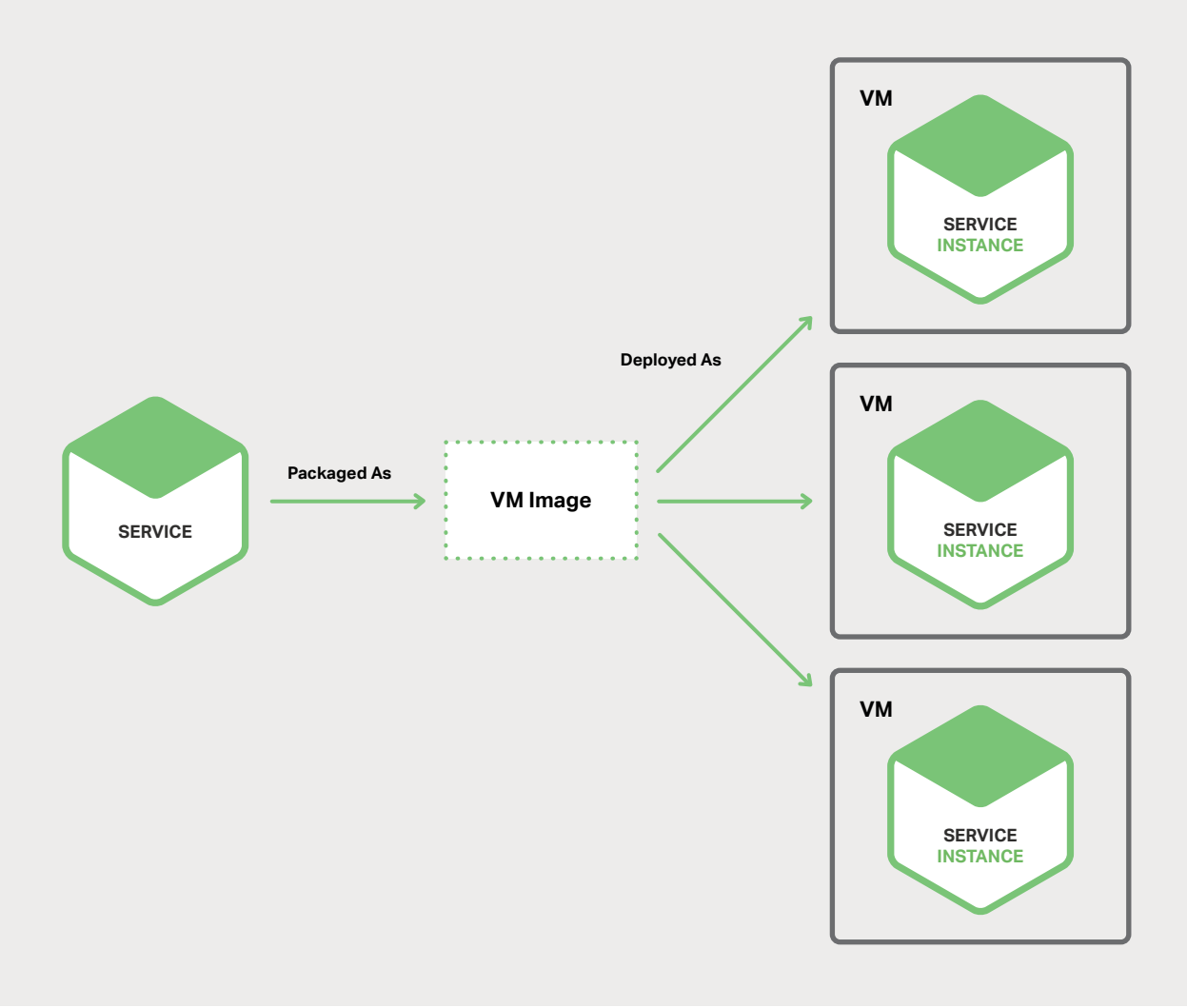


Figure 6-2. Services can each live in their own virtual machine.

This is the primary approach used by Netflix to deploy its video streaming service. Netflix packages each of its services as an EC2 AMI using Aminator. Each running service instance is an EC2 instance.

There are a variety tools that you can use to build your own VMs. You can configure your continuous integration (CI) server (for example, Jenkins) to invoke Aminator to package your services as an EC2 AMI. Packer is another option for automated VM image creation. Unlike Aminator, it supports a variety of virtualization technologies including EC2, DigitalOcean, VirtualBox, and VMware.

The company Boxfuse has a compelling way to build VM images, which overcomes the drawbacks of VMs that I describe below. Boxfuse packages your Java application as a minimal VM image. These images are fast to build, boot quickly, and are more secure since they expose a limited attack surface.

The company CloudNative has the Bakery, a SaaS offering for creating EC2 AMIs. You can configure your CI server to invoke the Bakery after the tests for your microservice pass. The Bakery then packages your service as an AMI. Using a SaaS offering such as the Bakery means that you don't have to waste valuable time setting up the AMI creation infrastructure.

The Service Instance per Virtual Machine pattern has a number of benefits. A major benefit of VMs is that each service instance runs in complete isolation. It has a fixed amount of CPU and memory and can't steal resources from other services.

Another benefit of deploying your microservices as VMs is that you can leverage mature cloud infrastructure. Clouds such as AWS provide useful features such as load balancing and autoscaling.

Another great benefit of deploying your service as a VM is that it encapsulates your service's implementation technology. Once a service has been packaged as a VM it becomes a black box. The VM's management API becomes the API for deploying the service. Deployment becomes much simpler and more reliable.

The Service Instance per Virtual Machine pattern has some drawbacks, however. One drawback is less efficient resource utilization. Each service instance has the overhead of an entire VM, including the operating system. Moreover, in a typical public IaaS, VMs come in fixed sizes and it is possible that the VM will be underutilized.

Moreover, a public IaaS typically charges for VMs regardless of whether they are busy or idle. An IaaS such as AWS provides autoscaling, but it is difficult to react quickly to changes in demand. Consequently, you often have to overprovision VMs, which increases the cost of deployment.

Another downside of this approach is that deploying a new version of a service is usually slow. VM images are typically slow to build due to their size. Also, VMs are typically slow to instantiate, again because of their size. Also, an operating system typically takes some time to start up. Note, however, that this is not universally true, since lightweight VMs such as those built by Boxfuse exist.

Another drawback of the Service Instance per Virtual Machine pattern is that usually you (or someone else in your organization) are responsible for a lot of undifferentiated heavy lifting. Unless you use a tool such as Boxfuse that handles the overhead of building and managing the VMs, then it is your responsibility. This necessary but time-consuming activity distracts from your core business.

Let's now look at an alternative way to deploy microservices that is more lightweight, yet still has many of the benefits of VMs.

Service Instance per Container Pattern

When you use the Service Instance per Container pattern, each service instance runs in its own container. Containers are a virtualization mechanism at the operating system level. A container consists of one or more processes running in a sandbox. From the perspective of the processes, they have their own port namespace and root filesystem. You can limit a container's memory and CPU resources. Some container implementations also have I/O rate limiting. Examples of container technologies include Docker and Solaris Zones.

Figure 6-3 shows the structure of this pattern:

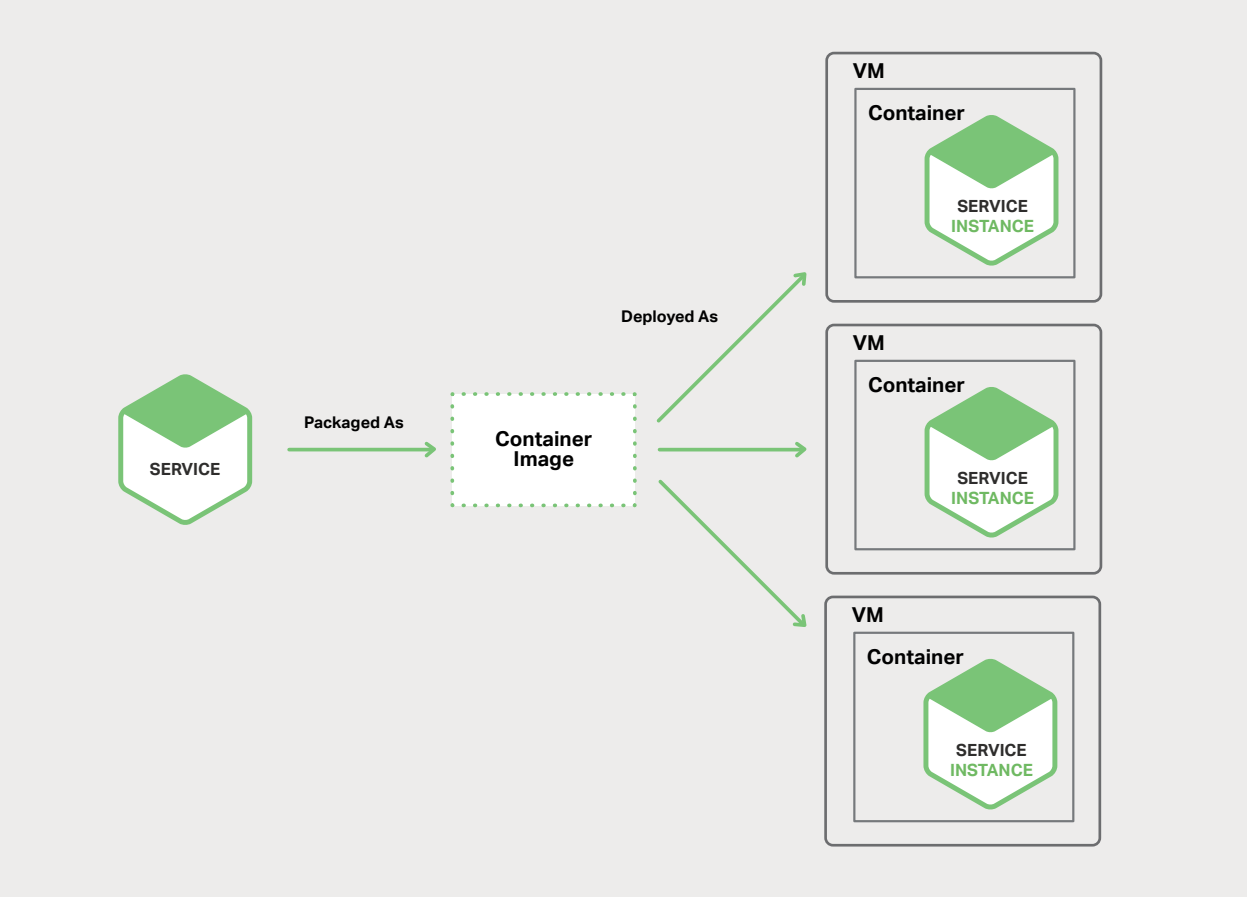


Figure 6-3. Services can each live in their own container.

To use this pattern, you package your service as a container image. A container image is a filesystem image consisting of the applications and libraries required to run the service. Some container images consist of a complete Linux root filesystem. Others are more lightweight. To deploy a Java service, for example, you build a container image containing the Java runtime, perhaps an Apache Tomcat server, and your compiled Java application.

Once you have packaged your service as a container image, you then launch one or more containers. You usually run multiple containers on each physical or virtual host. You might use a cluster manager such as Kubernetes or Marathon to manage your containers. A cluster manager treats the hosts as a pool of resources. It decides where to place each container based on the resources required by the container and resources available on each host.

The Service Instance per Container pattern has both benefits and drawbacks. The benefits of containers are similar to those of VMs. They isolate your service instances from each other. You can easily monitor the resources consumed by each container. Also, like VMs, containers encapsulate the technology used to implement your services. The container management API also serves as the API for managing your services.

However, unlike VMs, containers are a lightweight technology. Container images are typically very fast to build. For example, on my laptop it takes as little as 5 seconds to package a Spring Boot application as a Docker container. Containers also start very quickly, since there is no lengthy OS boot mechanism. When a container starts, what runs is the service.

There are some drawbacks to using containers. While container infrastructure is rapidly maturing, it is not as mature as the infrastructure for VMs. Also, containers are not as secure as VMs, since the containers share the kernel of the host OS with one another.

Another drawback of containers is that you are responsible for the undifferentiated heavy lifting of administering the container images. Also, unless you are using a hosted container solution such as Google Container Engine or Amazon EC2 Container Service (ECS), then you must administer the container infrastructure and possibly the VM infrastructure that it runs on.

Also, containers are often deployed on an infrastructure that has per-VM pricing. Consequently, as described earlier, you will likely incur the extra cost of overprovisioning VMs in order to handle spikes in load.

Interestingly, the distinction between containers and VMs is likely to blur. As mentioned earlier, Boxfuse VMs are fast to build and start. The Clear Containers project aims to create lightweight VMs. There is also growing interest in unikernels. Docker, Inc acquired Unikernel Systems in early 2016.

There is also the newer and increasingly popular concept of server-less deployment, which is an approach that sidesteps the issue of having to choose between deploying services in containers or VMs. Let's look at that next.

Serverless Deployment

AWS Lambda is an example of serverless deployment technology. It supports Java, Node.js, and Python services. To deploy a microservice, you package it as a ZIP file and upload it to AWS Lambda. You also supply metadata, which among other things specifies the name of the function that is invoked to handle a request (a.k.a. an event). AWS Lambda automatically runs enough instances of your microservice to handle requests. You are simply billed for each request based on the time taken and the memory consumed. Of course, the devil is in the details, and you will see shortly that AWS Lambda has limitations. But the notion that neither you as a developer, nor anyone in your organization, need worry about any aspect of servers, virtual machines, or containers is incredibly appealing.

A *Lambda function* is a stateless service. It typically handles requests by invoking AWS services. For example, a Lambda function that is invoked when an image is uploaded to an S3 bucket could insert an item into a DynamoDB images table and publish a message to a Kinesis stream to trigger image processing. A Lambda function can also invoke third-party web services.

There are four ways to invoke a Lambda function:

- Directly, using a web service request
- Automatically, in response to an event generated by an AWS service such as S3, DynamoDB, Kinesis, or Simple Email Service
- Automatically, via an AWS API Gateway to handle HTTP requests from clients of the application
- Periodically, according to a `cron`-like schedule

As you can see, AWS Lambda is a convenient way to deploy microservices. The request-based pricing means that you only pay for the work that your services actually perform. Also, because you are not responsible for the IT infrastructure, you can focus on developing your application.

There are, however, some significant limitations. Lambda functions are not intended to be used to deploy long-running services, such as a service that consumes messages from a third-party message broker. Requests must complete within 300 seconds. Services must be stateless, since in theory AWS Lambda might run a separate instance for each request. They must be written in one of the supported languages. Services must also start quickly; otherwise, they might be timed out and terminated.

Summary

Deploying a microservices application is challenging. You may have tens or even hundreds of services written in a variety of languages and frameworks. Each one is a mini-application with its own specific deployment, resource, scaling, and monitoring requirements. There are several microservice deployment patterns, including Service Instance per Virtual Machine and Service Instance per Container. Another intriguing option for deploying microservices is AWS Lambda, a serverless approach. In the next and final chapter of this ebook, we will look at how to migrate a monolithic application to a microservices architecture.

Microservices in Action: Deploying Microservices Across Varying Hosts with NGINX

by Floyd Smith

NGINX has a lot of advantages for various types of deployment – whether for monolithic applications, microservices apps, or hybrid apps (as described in the next chapter). With NGINX, you can abstract intelligence out of different deployment environments and into NGINX. There are many app capabilities that work differently if you use tools that are specific to different deployment environments, but that work the same way across all environments if you use NGINX.

This characteristic also opens up a second specific advantage for NGINX and NGINX Plus: the ability to scale an app by running it in multiple deployment environments *at the same time*. Let's say you have on-premise servers that you own and manage, but your app usage is growing and you anticipate spikes beyond what those servers can handle. Instead of buying, provisioning, and keeping additional servers warm "just in case", if you've "gone NGINX", you have a powerful alternative: scale into the cloud – for instance, scale onto AWS. That is, handle traffic on your on-premise servers until capacity is reached, then spin up additional microservice instances in the cloud as needed.

This is just one example of the flexibility that a move to NGINX makes possible. Maintaining separate testing and deployment environments, switching the infrastructure of your environments, and managing a portfolio of apps across all kinds of environments all become much more realistic and achievable.

The NGINX Microservices Reference Architecture is explicitly designed to support this kind of flexible deployment, with use of containers during development and deployment as an assumption. Consider a move to containers, if you're not there already, and to NGINX or NGINX Plus to ease your move to microservices and to future-proof your apps, development and deployment flexibility, and personnel.

7 Refactoring a Monolith into Microservices

This is the seventh and final chapter in this ebook about building applications with microservices. Chapter 1 introduces the Microservice Architecture pattern and discusses the benefits and drawbacks of using microservices. The subsequent chapters discuss different aspects of the microservices architecture: using an API Gateway, inter-process communication, service discovery, event-driven data management, and deploying microservices. In this chapter, we look at strategies for migrating a monolithic application to microservices.

I hope that this ebook has given you a good understanding of the microservices architecture, its benefits and drawbacks, and when to use it. Perhaps the microservices architecture is a good fit for your organization.

However, there is fairly good chance you are working on a large, complex monolithic application. Your daily experience of developing and deploying your application is slow and painful. Microservices seem like a distant nirvana. Fortunately, there are strategies that you can use to escape from the monolithic hell. In this article, I describe how to incrementally refactor a monolithic application into a set of microservices.

Overview of Refactoring to Microservices

The process of transforming a monolithic application into microservices is a form of application modernization. That is something that developers have been doing for decades. As a result, there are some ideas that we can reuse when refactoring an application into microservices.

One strategy not to use is the "Big Bang" rewrite. That is when you focus all of your development efforts on building a new microservices-based application from scratch. Although it sounds appealing, it is extremely risky and will likely end in failure. As Martin Fowler reportedly said, "the only thing a Big Bang rewrite guarantees is a Big Bang!"

Instead of a Big Bang rewrite, you should incrementally refactor your monolithic application. You gradually add new functionality, and create extensions of existing functionality, in the form of microservices – modifying your monolithic application in a complementary fashion, and running the microservices and the modified monolith in tandem. Over time, the amount of functionality implemented by the monolithic application shrinks, until either it disappears entirely or it becomes just another microservice. This strategy is akin to servicing your car while driving down the highway at 70 mph – challenging, but far less risky than attempting a Big Bang rewrite.

Martin Fowler refers to this application modernization strategy as the Strangler Application. The name comes from the strangler vine (a.k.a. strangler fig) that is found in rainforests. A strangler vine grows around a tree in order to reach the sunlight above the forest canopy. Sometimes, the tree dies, leaving a tree-shaped vine. Application modernization follows the same pattern. We will build a new application consisting of microservices around the legacy application, which will shrink and perhaps, eventually, die.

Let's look at different strategies for doing this.

Strategy #1 – Stop Digging

The Law of Holes says that whenever you are in a hole you should stop digging. This is great advice to follow when your monolithic application has become unmanageable. In other words, you should stop making the monolith bigger. This means that when you are implementing new functionality you should not add more code to the monolith. Instead, the big idea with this strategy is to put that new code in a standalone microservice.

Figure 7-1 shows the system architecture after applying this approach.

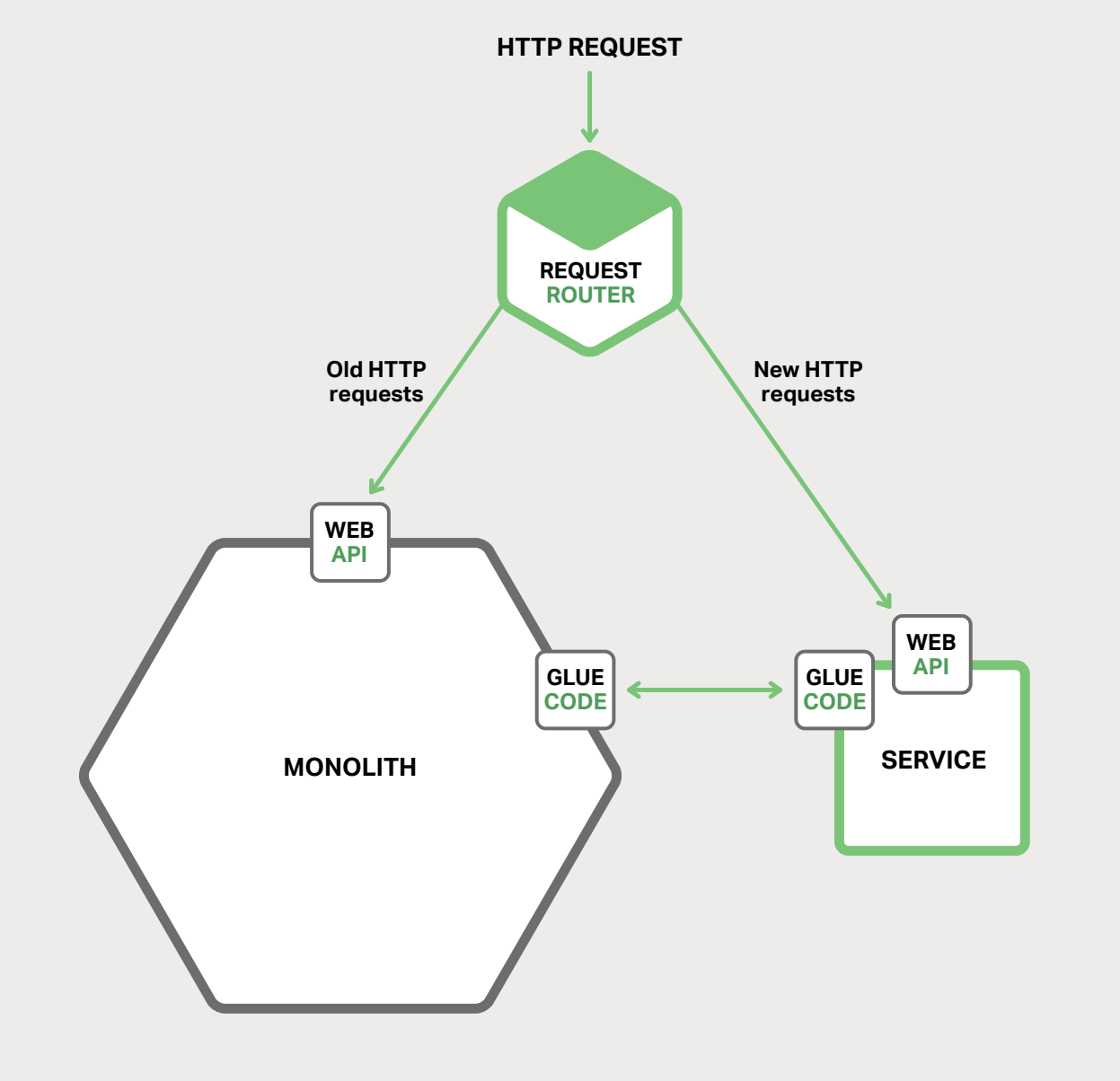


Figure 7-1. Implementing new functionality as a separate service instead of adding a module to the monolith.

As well as the new service and the legacy monolith, there are two other components. The first is a request router, which handles incoming (HTTP) requests. It is similar to the API gateway described in Chapter 2. The router sends requests corresponding to new functionality to the new service. It routes legacy requests to the monolith.

The other component is the glue code, which integrates the service with the monolith. A service rarely exists in isolation and often needs to access data owned by the monolith. The glue code, which resides in either the monolith, the service, or both, is responsible for the data integration. The service uses the glue code to read and write data owned by the monolith.

There are three strategies that a service can use to access the monolith's data:

- Invoke a remote API provided by the monolith
- Access the monolith's database directly
- Maintain its own copy of the data, which is synchronized with the monolith's database

The glue code is sometimes called an *anti-corruption layer*. That is because the glue code prevents the service, which has its own pristine domain model, from being polluted by concepts from the legacy monolith's domain model. The glue code translates between the two different models. The term anti-corruption layer first appeared in the must-read book *Domain Driven Design* by Eric Evans and was then refined in a white paper. Developing an anti-corruption layer can be a non-trivial undertaking. But it is essential to create one if you want to grow your way out of monolithic hell.

Implementing new functionality as a lightweight service has a couple of benefits. It prevents the monolith from becoming even more unmanageable. The service can be developed, deployed, and scaled independently of the monolith. You experience the benefits of the microservice architecture for each new service that you create.

However, this approach does nothing to address the problems with the monolith. To fix those problems you need to break up the monolith. Let's look at strategies for doing that.

Strategy #2 – Split Frontend and Backend

A strategy that shrinks the monolithic application is to split the presentation layer from the business logic and data access layers. A typical enterprise application consists of at least three different types of components:

- Presentation layer – Components that handle HTTP requests and implement either a (REST) API or an HTML-based web UI. In an application that has a sophisticated user interface, the presentation tier is often a substantial body of code.
- Business logic layer – Components that are the core of the application and implement the business rules.
- Data-access layer – Components that access infrastructure components, such as databases and message brokers.

There is usually a clean separation between the presentation logic on one side and the business and data-access logic on the other. The business tier has a coarse-grained API consisting of one or more facades, which encapsulate business-logic components. This API is a natural seam along which you can split the monolith into two smaller applications. One application contains the presentation layer. The other application contains the business and data-access logic. After the split, the presentation logic application makes remote calls to the business logic application.

Figure 7-2 shows the architecture before and after the refactoring.

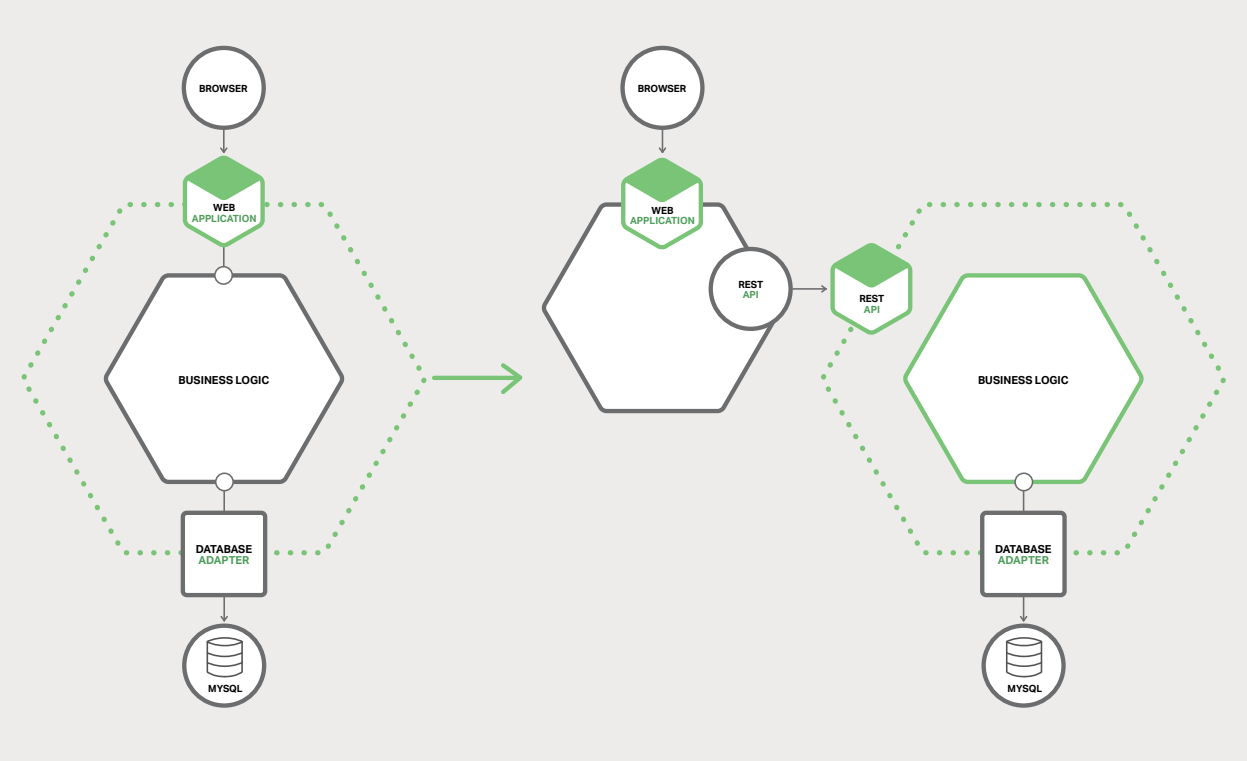


Figure 7-2. Refactoring an existing app.

Splitting a monolith in this way has two main benefits. It enables you to develop, deploy, and scale the two applications independently of one another. In particular, it allows the presentation-layer developers to iterate rapidly on the user interface and easily perform A|B testing, for example. Another benefit of this approach is that it exposes a remote API that can be called by the microservices that you develop.

This strategy, however, is only a partial solution. It is very likely that one or both of the two applications will be an unmanageable monolith. You need to use the third strategy to eliminate the remaining monolith or monoliths.

Strategy #3 – Extract Services

The third refactoring strategy is to turn existing modules within the monolith into standalone microservices. Each time you extract a module and turn it into a service, the monolith shrinks. Once you have converted enough modules, the monolith will cease to be a problem. Either it disappears entirely or it becomes small enough that it is just another service.

Prioritizing Which Modules to Convert into Services

A large, complex monolithic application consists of tens or hundreds of modules, all of which are candidates for extraction. Figuring out which modules to convert first is often challenging. A good approach is to start with a few modules that are easy to extract. This will give you experience with microservices in general and the extraction process in particular. After that, you should extract those modules that will give you the greatest benefit.

Converting a module into a service is typically time consuming. You want to rank modules by the benefit you will receive. It is usually beneficial to extract modules that change frequently. Once you have converted a module into a service, you can develop and deploy it independently of the monolith, which will accelerate development.

It is also beneficial to extract modules that have resource requirements significantly different from those of the rest of the monolith. It is useful, for example, to turn a module that has an in-memory database into a service, which can then be deployed on hosts, whether bare metal servers, VMs, or cloud instances, with large amounts of memory. Similarly, it can be worthwhile to extract modules that implement computationally expensive algorithms, since the service can then be deployed on hosts with lots of CPUs. By turning modules with particular resource requirements into services, you can make your application much easier and less expensive to scale.

When figuring out which modules to extract, it is useful to look for existing coarse-grained boundaries (a.k.a seams). They make it easier and cheaper to turn modules into services. An example of such a boundary is a module that only communicates with the rest of the application via asynchronous messages. It can be relatively cheap and easy to turn that module into a microservice.

How to Extract a Module

The first step of extracting a module is to define a coarse-grained interface between the module and the monolith. It is mostly likely a bidirectional API, since the monolith will need data owned by the service and vice versa. It is often challenging to implement such an API because of the tangled dependencies and fine-grained interaction patterns between the module and the rest of the application. Business logic implemented using the Domain Model pattern is especially challenging to refactor because of numerous associations between domain model classes. You will often need to make significant code changes to break these dependencies. Figure 7-3 shows the refactoring.

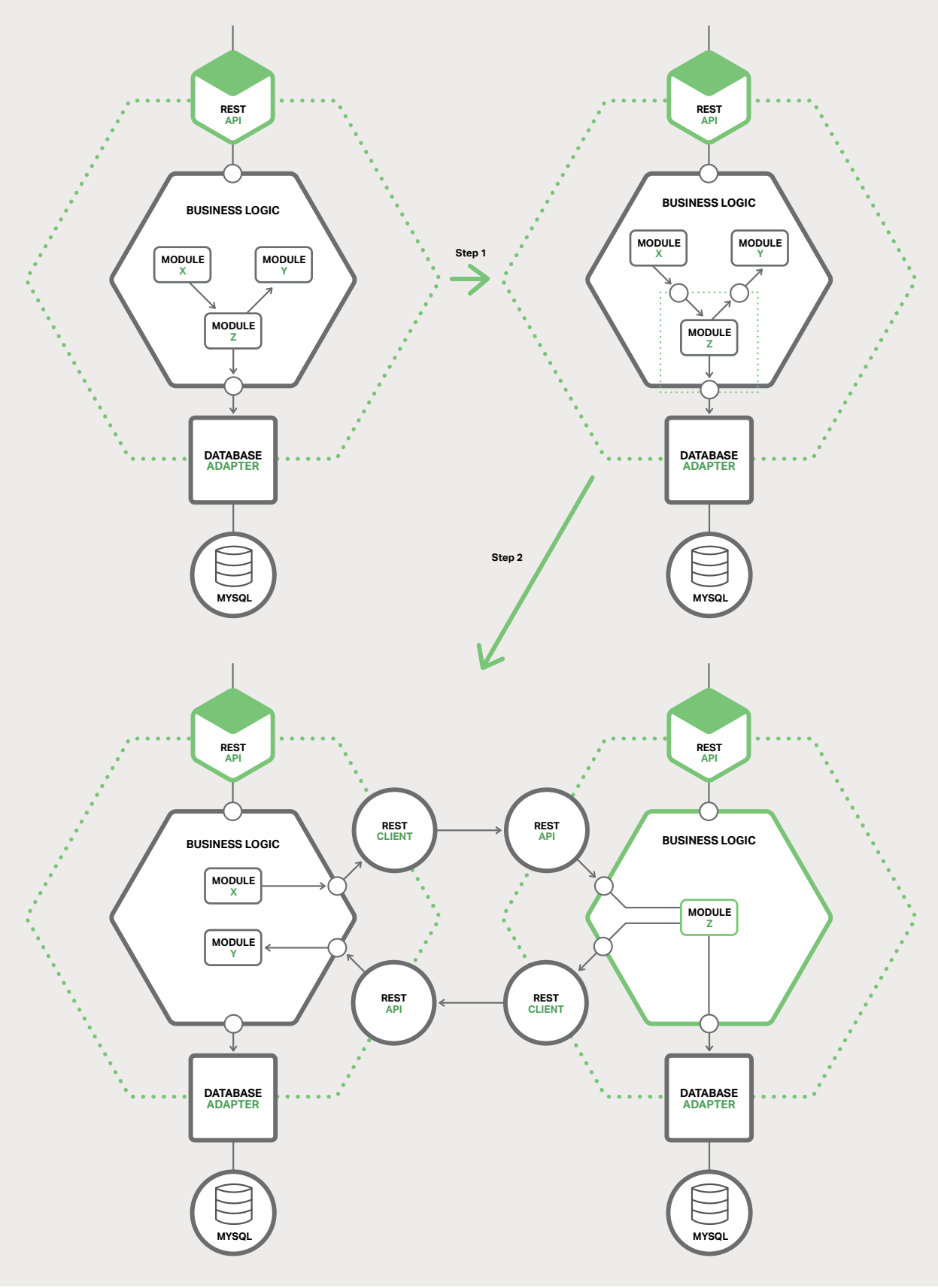


Figure 7-3. A module from a monolith can become a microservice.

Once you implement the coarse-grained interface, you then turn the module into a free-standing service. To do that, you must write code to enable the monolith and the service to communicate through an API that uses an inter-process communication (IPC) mechanism. Figure 7-3 shows the architecture before, during, and after the refactoring.

In this example, Module Z is the candidate module to extract. Its components are used by Module X and it uses Module Y. The first refactoring step is to define a pair of coarse-grained APIs. The first interface is an inbound interface that is used by Module X to invoke Module Z. The second is an outbound interface used by Module Z to invoke Module Y.

The second refactoring step turns the module into a standalone service. The inbound and outbound interfaces are implemented by code that uses an IPC mechanism. You will most likely need to build the service by combining Module Z with a Microservice Chassis framework that handles cross-cutting concerns such as service discovery.

Once you have extracted a module, you have yet another service that can be developed, deployed, and scaled independently of the monolith and any other services. You can even rewrite the service from scratch; in this case, the API code that integrates the service with the monolith becomes an anti-corruption layer that translates between the two domain models. Each time you extract a service, you take another step in the direction of microservices. Over time, the monolith will shrink and you will have an increasing number of microservices.

Summary

The process of migrating an existing application into microservices is a form of application modernization. You should not move to microservices by rewriting your application from scratch. Instead, you should incrementally refactor your application into a set of microservices. There are three strategies you can use: implementing new functionality as microservices; splitting the presentation components from the business and data access components; and converting existing modules in the monolith into services. Over time the number of microservices will grow, and the agility and velocity of your development team will increase.

Microservices in Action: Taming a Monolith with NGINX

by Floyd Smith

As this chapter describes, converting a monolith to microservices is likely to be a slow and challenging process, yet one with many benefits. With NGINX, you can begin to get some of the benefits of microservices before you actually begin the conversion process.

You can buy a lot of time for the move to microservices by "dropping NGINX in front of" your existing monolithic application. Here's a brief description of the benefits as they relate to microservices:

- **Better support for microservices** – As mentioned in the sidebar for Chapter 5, NGINX, and NGINX Plus in particular, have capabilities that help enable the development of microservices-based apps. As you begin to re-design your monolithic application, your microservices will perform better and be easier to manage due to the capabilities in NGINX.
- **Functional abstraction across environments** – Moving capabilities onto NGINX as a reverse proxy server reduces the number of things that will vary when you deploy across new environments, from servers you manage to various flavors of public, private, and hybrid clouds. This complements and extends the flexibility inherent to microservices.
- **Availability of the NGINX Microservices Reference Architecture** – As you move to NGINX, you can borrow from the NGINX Microservices Reference Architecture, both to define the ultimate structure of your app after the move to microservices, and to use parts of the MRA as needed for each new microservice you create.

To sum up, implementing NGINX as a first step in your transition takes the pressure off your monolithic application, makes it much easier to attain all of the benefits of microservices, and gives you models for use in making the transition. You can learn more about the MRA and get a free trial of NGINX Plus today.

Resources for Microservices and NGINX

by Floyd Smith

The NGINX website is already a valued resource for people seeking to learn about microservices and implement them in their organizations. From introductory descriptions, such as the first chapter of this ebook, to advanced resources such as the Fabric Model of the NGINX Microsoft Reference Architecture, there's a graduate seminar-level course in microservices available at **https://www.nginx.com**.

Here are a few tips and tricks, and a few key resources, for getting started on your journey with NGINX and microservices:

- **Site search and web search.** The best way to search the NGINX website for microservices material is to use site-specific search in Google:
  - **site:nginx.com *topic*** to search the NGINX website.
  - **site:nginx.com/blog *topic*** to search the NGINX blog. All blog posts are tagged, so once you find a topic you want to follow up on, just click the tag to see all relevant posts. Authors are linked to all their articles as well.
  - Search for ***topic*** **nginx** to find content relevant to both NGINX and your topic of choice on the Web as a whole – there's a lot of great stuff out there. DigitalOcean may be the best external place to start.
- **General NGINX resources.** Here are links to different types of content on the NGINX site:
  - Blog posts. Once you find a post on microservices, click the microservices tag to see all such posts.
  - Webinars. Click the Microservices filter to see microservices-relevant webinars.
  - White papers, reports, and ebooks. Use site search on this part of the site, as described above, to find resources relating specifically to microservices and other topics of your choice.
  - NGINX YouTube channel. NGINX has dozens of videos, including all the presentations from several years of our annual conference. Many of these videos have been converted into blog posts if you prefer reading to watching; search for the name of the speaker in the NGINX blog.

- **Specific resources.** Microservices is the single most popular, and best-covered, topic on the NGINX website. Here are a few "best of the best" resources to get you started.
  - This ebook as a blog post series. Look in the NGINX blog to find the Chris Richardson blog posts that were (lightly) adapted to form the seven chapters of this ebook.
  - Building Microservices ebook. A free download of an O'Reilly animal book on microservices. Need we say more?
  - Microservices at Netflix. Netflix is a leader in implementing microservices, moving to the cloud, and making their efforts available as open source – all based on NGINX, of course.
  - Why NGINX for Containers and Microservices? The inimitable Owen Garrett on a topic dear to our hearts.
  - Implementing Microservices. A fresh take on the topic of this ebook, emphasizing the four-tier architecture.
  - Introducing the NGINX Microservices Reference Architecture. Professional services maven Chris Stetson introduces the MRA.